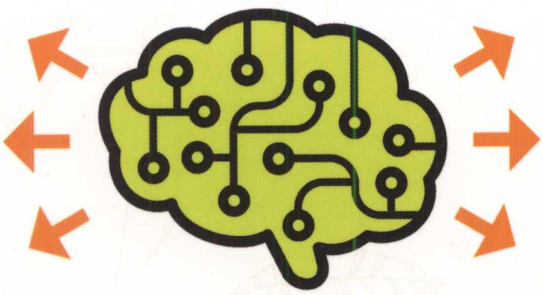
Broadview®
www.broadview.com.cn

算法笔记

Algorithm Notes

刁瑞 谢妍 / 著

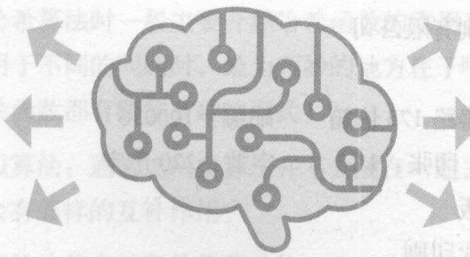


中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

算法笔记

Algorithm Notes

刁瑞 谢妍 / 著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

本书介绍了若干常见算法，既包括排序、哈希等基础算法，也包括无约束优化、插值与拟合等数值计算方法。本书在介绍算法的同时，结合了作者自己对数学背景、应用场景的理解，便于读者把握算法的核心思想。本书尽可能地避开了以应试为导向的灌输式讲解，力求引起读者的兴趣并扩大其视野，例如在介绍哈希时，讲解了如何将哈希的算法思想运用于相似性搜索、负载均衡等多个实际问题中；又如在介绍高斯消去法时，讲解了相关的数学理论及编程实现上的具体技巧，并将其运用于对大规模稀疏线性方程组的求解，等等。

本书面向有一定高等数学、编程语言基础及对算法有初步了解的读者，包括高等院校的学生、程序员、算法分析人员及设计人员等，旨在帮助读者进一步学习算法，理解与算法相关的理论基础和应用实例。

图书在版编目（CIP）数据

算法笔记 / 刁瑞，谢妍著. —北京：电子工业出版社，2016.7
ISBN 978-7-121-28671-1
I. ① 算… II. ① 刁… ② 谢… III. ① 电子计算机－算法理论 IV. ① TP301.6

中国版本图书馆 CIP 数据核字（2016）第 089324 号

策划编辑：张国霞
责任编辑：徐津平
印　　刷：北京中新伟业印刷有限公司
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　印张：11.5　　字数：220 千字
版　　次：2016 年 7 月第 1 版
印　　次：2017 年 1 月第 3 次印刷
印　　数：5001～6000 册　　定价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819　faq@phei.com.cn。

前言

本书取名“算法笔记”，主要源自作者在中国科学院读书期间学习算法时的体会，可以作为现有算法教科书的补充。本书讨论了计算机算法相关的若干话题，在介绍算法的同时结合了作者自己对数学背景、应用场景的理解，便于读者把握算法的核心思想。阅读本书需要有一定的数学基础和算法基础。

许多经典的算法教科书都详尽地介绍了算法的各个知识点，但在覆盖面广的同时难免会忽略许多细节问题。例如，哪些算法真正值得运用到实际问题中，算法有哪些变种值得我们了解，算法背后有哪些数学理论支撑，等等。

本书共包括 8 章。各章中除了讲解基本知识，还回答了许多相关的有趣问题。

- 排序：排序算法有很多种，在比较流行的编程语言中都有提供排序算法的库函数，直接调用这些库函数会非常简单。但它们所使用的算法为何有效，这些算法与一些经典的排序算法又有什么区别？
- 哈希：在讲解哈希算法时一般主要介绍哈希函数的作用及哈希表的不同实现方法。但将哈希函数运用于不同的问题时，最为巧妙的地方在于哈希函数的设计。对于不同领域的问题，哈希函数都有哪些有趣的形式？
- 动态规划与近似算法：通常这两类算法并不会放在一起去探讨。在面对不同复杂性的问题时，它们会有怎样的互补作用？
- 高斯消去法：算法的基本过程是很简单的，但在实际使用中远远没有那么简单。如何保持计算的稳定性？如何解决稀疏矩阵的计算效率问题？
- 图论与线性规划：图论中的许多问题都可以用线性规划去解决。图论中的一些经典结论实质上也可以用线性规划的相关定理去解释。线性规划作为一个更一般的工具，如何用于处理图论问题？

- 无约束优化：无约束优化主要用于求解函数的最大值或最小值的问题。常用的这些方法为何有效？它们之间的差别在哪里？
- 迭代法：常见的迭代算法都有哪些？它们为什么有效？
- 插值与拟合：插值与拟合的思想是什么？有什么异同？如何运用于图像处理？

读者可以发现，本书不仅指出了哪些算法可以解决问题，还指出了哪些算法可以更好地解决问题。这有助于我们对算法的深入理解。

由于作者水平有限，书中难免有错误和不足之处，欢迎读者批评和指正。

刁瑞、谢妍

2016 年 7 月

目录

第1章 排序

将一堆杂乱无章的元素按照某种规则有序排列的过程叫作“排序”。排序是非常基础和重要的一类算法，有着广泛的理论基础和实践需求。比如，英文字典中的单词是按照字母序排列的，我们可以方便、快速地查找到某个单词的中文翻译；再比如，将我们中学期间可怕的成绩排行榜按照考分倒序排列后，班主任会挑出前几名优秀生进行表扬，然后挑出排在最后几名的同学着重辅导。本章我们将介绍和讨论一些实用的排序算法，按照是否基于“比较”操作，将这些算法分为比较排序和非比较排序两大类。

1.1 比较排序

大多数排序算法都属于比较排序。在比较排序中，待排序对象可以是任意数据类型，我们只需知道如何比较两个对象的大小。我们要研究的核心问题可以描述如下。

定义 1.1（基于比较的排序） 给定一个包含 n 个对象的待排序序列 $a_1, a_2, \cdots, a_n$。假设我们知道如何比较其中任意两个对象的大小关系，以及如何对这一序列排序。

基于比较的排序需要明确比较两个对象的大小关系的规则。如果是对整数、实数等对象进行排序，则大小关系的定义非常明确。对于自定义对象，其比较规则应当满足两个性质。

- 传递性：如果 $a \leqslant b$，$b \leqslant c$，则一定有 $a \leqslant c$。
- 全序性：对任意 a 和 b，或者 $a \leqslant b$，或者 $b \leqslant a$。

有可能存在 $a \leqslant b$ 和 $b \leqslant a$ 同时成立的情况，这时任何一个都可以排在前面。但有时我

们也会要求排序是稳定的，即在 $a \leqslant b$ 和 $b \leqslant a$ 同时成立时，它们需要保持初始序列中的前后顺序。

有一种很巧妙的得到比较排序的算法复杂度下界的办法。n 个对象的排列一共有 $n!$ 种，在最坏的情况下只有其中某一个排列是排序结果。一个排序算法需要足够多的信息才能确定这个结果。如果对任何待排序序列都能经过至多 $f(n)$ 次比较后结束，那么它至多可以区分 $2^{f(n)}$ 种情况，因为每进行一次比较，只可能有两种结论，所以 $2^{f(n)} \geqslant n!$，即 $f(n) \geqslant \log_2(n!)$。根据斯特灵公式（Stirling's formula），$\log_2(n!)$ 实际上就是 $\Omega(n \log_2 n)$。许多知名的排序算法都达到了这个下界。

本节将主要介绍一些在实际测试中表现较好且得到广泛使用的排序算法。另外，除了基本的排序问题，我们也会考虑一些特殊的排序问题，其中主要包括两个问题：合并多个有序列，以及前 k 小数。

定义 1.2（合并多个有序列） 将 n 个有序列合并成一个有序列，这 n 个有序列的长度分别为 $m_1, m_2, \cdots, m_n$。

合并多个有序列的问题有许多应用。例如我们需要检索一个数据库，而检索结果只需满足多个条件中的某一个。这时我们可以先进行单一条件检索，然后将这些检索结果进行合并。而单一条件检索往往是可以借助索引快速完成的，因此合并检索结果的效率更为重要。一个简单的解法是直接使用排序算法对这 n 个有序列整体做一次排序，时间复杂度为 $O((\sum_{i=1}^{n} m_i) \log(\sum_{i=1}^{n} m_i))$。

定义 1.3（前 k 小数） 给定一个包含 n 个对象的序列，找出前 k 个最小的数。

前 k 小数的问题也有许多应用。例如，当检索结果太多时，我们可以根据某个评价标准只列出最值得展示的 k 个结果作为参考，这样可以避免对浏览造成负担。

1.1.1 梳排序

冒泡排序是许多人接触的第 1 个排序算法，虽然并不实用，却最为简单。本节我们要介绍的梳排序（Comb Sort）可以作为冒泡排序的一种改进，虽然它并没有很好的理论结果，但实际效果非常好。冒泡排序的算法流程如下。

算法 1 冒泡排序

1: **for** $i = 1, 2, \ldots, n-1$ **do**
2: **for** $j = 1, 2, \ldots, n-i$ **do**

```
    if a_j > a_{j+1} then
      交换 a_j 和 a_{j+1}。
    end if
  end for
end for
```

如果对冒泡排序的理解不正确，则很容易弄错循环顺序。冒泡排序的特点是调整相邻两个对象的位置，每进行一次内循环，就可以将最大值调整到最后，这样以后就不必再考虑它了。在进行 $n-1$ 次内循环后，就得到了完整的有序列。冒泡排序的时间复杂度是 $O(n^2)$，虽然在时间上并不占优势，但代码非常简洁，实现难度很低。

与冒泡排序相比，更加实用的梳排序反而知名度低一些。梳排序同样具有代码简洁的特点。梳排序的算法流程如下。

算法 2 梳排序

```
j ← n，s ← 1.3，flag ← false。
while j > 1 或者 flag = true do
  i ← 0，j ← max{⌊j/s⌋, 1}，flag ← false。
  while i + j ⩽ n do
    if a_i > a_{i+j} then
      交换 a_i 和 a_{i+j}。
      flag ← true。
    end if
    i ← i + 1。
  end while
end while
```

我们来解释一下这个算法。注意到在算法中包含一个变量 j，当 $j=1$ 的时候，我们可以验证梳排序算法的行为和冒泡排序是完全一致的，即调整相邻两个对象的位置。梳排序是从很大的 j 开始的，逐渐缩小 j 直到 1。在 j 变成 1 之前，它都和冒泡排序不一样。也就是说，梳排序相当于在冒泡排序之前添加了一些排序工作。

为了简单起见，我们讨论 $n=8, j=3$ 的情况。在算法内层循环中，我们只会比较 a_i 和 a_{i+3}，所以相当于我们将 8 个数划分为 3 个不同的组。分组方式如表 1.1所示。我们只对同组的数进行一轮冒泡，对不同组的数不会进行比较。这个分组的思想和归并排序有些类似。整个梳排序就是在通过不断减小 j 来减少分组数量，而对于固定的 j，我们只对同组的

数进行一轮冒泡。可以想象，当 j 很大的时候，组很多，每组的数很少，工作量也就很少。因此在退化成冒泡排序之前，时间复杂度不会太高。我们可以大致估计，在 $j > 1$ 时，即退化成冒泡排序之前，梳排序进行了多少次比较。对于一个固定的 j，我们将 n 个数分为 j 组，每组中的数不超过 n/j 个，因此总的比较次数明显不会达到 n 次。而 j 每次除以 1.3，从 n 一直降到 1，因此我们对 j 的不同选择不会超过 $\log_{1.3} n$，所以总的比较次数的上界是 $n \log_{1.3} n$，这是低于冒泡排序的复杂度的。也就是说梳排序在退化之前的计算量要低于冒泡排序。在实际计算时，通常在 $j = 1$ 之前，梳排序就已经完成了大部分排序工作，使得它的表现要远远好于冒泡排序。

表 1.1 梳排序的分组示例

序　列	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
分　组	A	B	C	A	B	C	A	B

当需要快速编写一个效率较高的排序算法时，梳排序不失为一个明智的选择。

1.1.2 堆排序

堆排序（Heapsort），顾名思义就是借助堆（Heap）这个数据结构来排序。堆的实现有很多种，但它们都支持常见的几种操作。

- 建堆（即初始化堆）。
- 查找最大值或查找最小值，一般只支持其中一种。用于查找最大值的堆叫作最大堆，用于查找最小值的堆叫作最小堆。
- 删除最大值或删除最小值。
- 插入（即插入一个新的元素到堆）。

这里我们可以只考虑最小堆。堆的最简单的一个实现就是二叉堆。假设堆中有 n 个对象，则二叉堆的几个操作的复杂度分别如下。

- 建堆：$O(n)$。
- 查找最小值：$O(1)$。
- 删除最小值：$O(\log n)$。
- 插入：$O(\log n)$。

堆排序的流程非常简单，就是直接利用堆的这几种操作。我们只需对全部待排序对象建堆，然后反复查找并删除最小值即可。这样我们就可以得到一个从小到大排序后的结果。

堆排序的思想可以用于解决多个有序列的合并问题。多个有序列的合并问题已经在前面的定义1.2中给出。很显然，合并后的最小值应当是这 n 个有序列的最小值中的某一个，因此我们可以将每个有序列中的第 1 个对象即最小值放入堆中，进行建堆。之后，我们从中查找并删除最小值。如果删除的最小值来自第 j 个有序列，则我们需要将第 j 个有序列的下一个尚未处理的对象放入堆中，始终保持在堆中包含每个有序列的第 1 个尚未处理的对象。建堆的时间复杂度为 $O(n)$，每一次查找并删除最小值的复杂度是 $O(\log n)$，总共需要查找并删除最小值的次数是 $\sum_{i=1}^{n} m_i$，每一次插入的复杂度是 $O(\log n)$，总共需要插入的次数是 $\sum_{i=1}^{n}(m_i-1)$。因此，总的时间复杂度是 $O(\sum_{i=1}^{n} m_i \log n)$。

堆排序的思想还可以用于查找前 k 个小数。我们直接使用堆排序就可以解决这个问题。唯一区别是，在堆排序找出前 k 个最小的数之后，就可以提前退出了。建堆的时间复杂度为 $O(n)$，每一次查找并删除最小值的复杂度是 $O(\log n)$，总共需要查找并删除最小值的次数是 k，因此总的时间复杂度是 $O(n + k\log n)$。

1.1.3 归并排序

归并排序使用了分治算法的思想，每次将序列分为两半并分别处理，最终合并成一个序列。

算法 3 归并排序（递归版）

1: 如果 $n \leqslant 1$，则不必排序，程序结束。
2: 递归对 $a_1, a_2, \cdots, a_{\lceil n/2 \rceil}$ 排序。
3: 递归对 $a_{\lceil n/2 \rceil+1}, a_{\lceil n/2 \rceil+2}, \cdots, a_n$ 排序。
4: 合并两个有序列，得到最终的排序结果。

归并排序的过程可以看作一棵二叉树。树的每个叶子节点对应序列中的一个对象，其他每个节点都对应由两个子节点合并得到的序列，根节点对应最终的排序结果。例如，我们要对 $6,5,3,1,7,2,4$ 从小到大排序，则归并排序的过程如图 1.1所示。

我们现在还需要解释的是如何合并两个有序列。很显然，我们可以比较这两个有序列的第 1 个数，然后将其中最小的数作为合并后的序列的第 1 个数，因为这个数一定是最小的数。对于两个有序列的其余的数，我们可以继续这样处理，选择下一个最小的数。

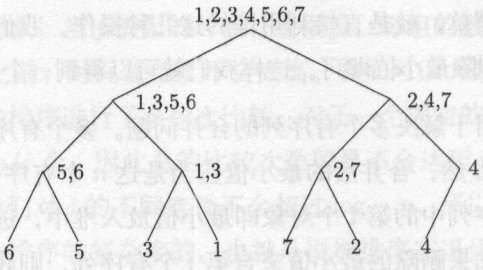


图 1.1 归并排序示例

算法 4 合并两个有序列

输入: 有序列 $x_1 \leqslant x_2 \leqslant \cdots \leqslant x_m$ 和 $y_1 \leqslant y_2 \leqslant \cdots \leqslant y_n$。

输出: 合并后的有序列 $z_1 \leqslant z_2 \leqslant \cdots \leqslant z_{m+n}$。

1: $i \leftarrow 1$, $j \leftarrow 1$。
2: **while** $i \leqslant m$ 且 $j \leqslant n$ **do**
3: **if** $x_i < y_j$ **then**
4: $z_{i+j-1} \leftarrow x_i$, $i \leftarrow i+1$。
5: **else**
6: $z_{i+j-1} \leftarrow y_j$, $j \leftarrow j+1$。
7: **end if**
8: **end while**
9: **if** $i \leqslant m$ **then**
10: $z_{k+n} \leftarrow x_k$, $\forall k = i, i+1, \ldots, m$。
11: **else**
12: $z_{k+m} \leftarrow y_k$, $\forall k = j, j+1, \ldots, n$。
13: **end if**

归并排序还可以写成非递归的形式。非递归的好处是节约了多次函数调用的资源消耗，但同时代码的可读性通常会比递归差。

算法 5 归并排序（非递归版）

1: $j \leftarrow 1$。
2: **while** $j < n$ **do**
3: $i \leftarrow 1$。
4: **while** $i + j - 1 < n$ **do**

5: 合并两个有序列 $a_i, a_{i+1}, \ldots, a_{i+j-1}$ 和 $a_{i+j}, a_{i+j+1}, \ldots, a_{\min\{i+2j-1,n\}}$。

6: $i \leftarrow i + 2j$。

7: **end while**

8: $j \leftarrow 2j$。

9: **end while**

算法5可以结合图 1.1来理解。它相当于在递归树上自底向上地进行处理。在递归树的同一层中，除了最后一个子序列，其余所有子序列的长度都是相同的，均为 j。因此，仅仅根据下标，我们就可以确定每个子序列的起始位置，从而判断出应该对哪些子序列进行两两合并。

在讲解堆排序时，我们讨论了如何合并多个有序列。归并排序的核心是合并两个有序列，其思想同样可以用于合并多个有序列。合并多个有序列看上去与合并两个有序列不同，但实际上归并排序也可以解决，其思路就是将有序列两两归并，这一过程仍然可以看作一棵树。但不同的是，树的平衡性可能会发生变化，从而影响计算效率。

先考虑一种简单的情况，使用类似归并排序的规则进行合并。图 1.2为 $n = 7$ 时的示意图，其中的每个节点处标注的是有序列的长度，算法结束时得到的有序列的长度就是根节点所标注的 $m_1 + m_2 + \cdots + m_7$。

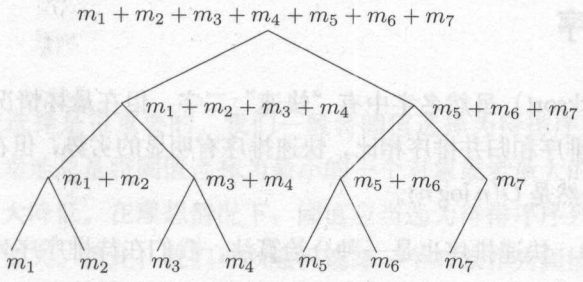


图 1.2 合并多个有序列的示例

很明显，这棵树的高度是 $O(\log n)$，而同一层上的合并操作的复杂度总和是 $O(\sum_{i=1}^{n} m_i)$，所以总的时间复杂度应当是 $O(\log n \sum_{i=1}^{n} m_i)$。这一复杂度与前面在使用堆时的复杂度是相同的。实际上，归并排序本身可以看作合并 n 个长度均为 1 的有序列，如果令 $m = 1$，则刚好得到归并排序的时间复杂度是 $O(n \log n)$。

当这些有序列的长度 $m_1, m_2, \cdots, m_n$ 之间的差异很大时，合并的次序可能会对计算时间有很大的影响。来看一个极端的例子，假设待合并的 n 个有序列的长度分别为 1, 1, 2, 4,

$8, 16, 32, \cdots, 2^{n-2}$。如果我们按照一般的合并方式，则时间复杂度是 $O(2^{n-1}\log n)$。如果每次选择当前最短的两个有序列进行合并，则第 1 次合并两个长度为 1 的有序列，第 2 次合并两个长度为 2 的有序列，第 3 次合并两个长度为 4 的有序列……第 $n-1$ 次合并两个长度为 2^{n-2} 的有序列。总的时间复杂度仅为 $O(2^n)$，明显低于一般的合并方式。

我们来看一下二叉树的形状是如何影响计算时间的。无论合并的次序如何，我们仍然可以将归并操作对应到二叉树，但是对这棵二叉树的具体形状可以进行调整。每个叶子节点总是可以对应一个初始的有序列，我们记第 i 个有序列所在叶子节点的深度为 h_i。如果我们仅考虑合并有序列这个操作，则结合二叉树，其计算量应当等于 $\sum_{i=1}^{n} m_i h_i$。让计算量达到最小，实际上和霍夫曼编码（Huffman Coding）的目标函数是完全一致的，因此我们可以借助霍夫曼编码的构造方法来选择序列的合并顺序，即每次选择当前最短的两个有序列进行合并。根据霍夫曼编码的理论可以证明，这样的策略使得所有合并操作达到最小的计算量。这样的思路很有意义但并不实用，因为合并两个在内存上并不相邻的有序列比较难于处理，我们不得不频繁开辟新的内存空间来存储合并结果。所以我们可以制定一个相对简单的策略，每次选择两个相邻的且总长度最小的子序列进行合并，这样也能达到较好的效果。我们可以使用堆来维护和查找总长度最小的两个相邻子序列。在后面将要介绍的 Timsort 中也面临类似的问题，其解决方案也非常巧妙。

1.1.4 快速排序

快速排序（Quicksort）虽然名字中有“快速”二字，但在最坏情况下的时间复杂度是 $O(n^2)$。与前面的堆排序和归并排序相比，快速排序有明显的劣势，但在平均意义下，快速排序的时间复杂度仍然是 $O(n\log n)$。

同归并排序一样，快速排序也是一种分治算法。我们在待排序序列中选择一个对象作为阈值，将小于它和大于它的分别放入两个子序列中，将相等的放入任意一个序列，然后分别对这两个子序列排序。这一流程很容易写成递归的形式。

算法 6 快速排序（递归版）

1: 选择一个阈值，将比阈值小的对象放入子序列 X 中，将比阈值大的对象放入子序列 Y 中。
2: 若子序列 X 包含超过 1 个对象，则递归对子序列 X 排序。
3: 若子序列 Y 包含超过 1 个对象，则递归对子序列 Y 排序。
4: 连接子序列 X, Y 得到最终排序后的序列。

快速排序实质上也对应一棵二叉树，这一点从算法描述中就有所体现。每选定一个阈值就可以将当前节点划分成两个子节点，左子树处理所有比阈值小的对象，右子树处理所有比阈值大的对象。

快速排序还有非递归版本。在非递归版本中就需要自己维护尚未完全处理好的子序列。我们记集合 S 中的每一个元素 (l,r) 代表 $a_l, a_{l+1}, \ldots, a_r$，是一个待排序子序列。

算法 7 快速排序（非递归版）

初始化集合 $S \leftarrow \{(1,n)\}$。
while S 不为空 **do**
  从 S 中取出一个元素 (l,r)。
  选择一个阈值，将比阈值小的对象调整位置至 $a_l, a_{l+1}, \ldots, a_{m-1}$，将比阈值大的对象调整位置至 $a_{m+1}, a_{m+2}, \ldots, a_r$。
  if $m-l>1$ **then**
    $S \leftarrow S \cup \{(l,m-1)\}$。
  end if
  if $r-m>1$ **then**
    $S \leftarrow S \cup \{(m+1,r)\}$。
  end if
end while

快速排序的阈值选择是很重要的。我们一般将阈值选择为待排序序列中的某个对象。可以想象，如果很不幸地总是将阈值选择为最小的一个对象或者最大的一个对象，则快速排序将退化，效率大大降低。在理想情况下，阈值应当选为待排序序列的中位数，但精确地找到中位数的代价太大。因此，我们可以随机选择一个对象作为阈值，这样就很难退化到极端情况，但是这样做有以下两个缺点。

- 随机数本身会带来一些问题。一方面，生成随机数会有一定的开销；另一方面，会导致计算结果的不一致性。所谓不一致性是指每次排序可能会产生不同的结果。
- 虽然很难退化到极端情况，但也很难达到最优情况。

有一种很有趣的解决方案是，采用待排序序列中的第 1 个对象、中间一个对象及最后一个对象的中位数作为阈值。这种解决方案完全避开了随机数，也不会退化到极端情况，而达到最优情况的概率有所增加。我们可以具体计算一下阈值刚好是中位数的概率。为了方便起见，我们假设 $n=2k+1$ 且有唯一的中位数，待排序序列的初始顺序是随机的，则三

个对象的中位数刚好是整个待排序序列的中位数的概率是

$$\frac{k^2}{n(n-1)(n-2)/6} = \frac{3k}{4k^2-1} \tag{1.1}$$

而随机选择阈值刚好是中位数的概率是

$$\frac{1}{n} = \frac{1}{2k+1} = \frac{2k-1}{4k^2-1} \tag{1.2}$$

可以看到，前者大约是后者的 1.5 倍。

我们之前用堆排序的思想给出了前 k 小数的求解思路。快速排序的思想同样可以用于求解前 k 小数。在快速排序中，我们每一次选定阈值并划分对象为两个子序列。这时很容易判断第 k 个小数在哪个子序列中。此后只需要对这个子序列进行排序，而不必再关心另一个子序列的顺序，这就减少了今后的工作量。在平均意义下，这样做的复杂度是多少呢？再次观察快速排序所对应的二叉树，我们相当于只处理了这棵树中的一条链。对一条链上的所有序列长度求和可知，其在平均意义下的时间复杂度为 $O(n)$。与堆排序中所介绍的方法相比，使用快速排序的思想计算前 k 小数会更快。与堆排序不同的是，快速排序并未对前 k 个最小的数进行排序。如果需要，则可以在找到前 k 小数后，再单独对它们进行一次排序。

1.1.5 内省排序

前面我们已经介绍了若干常用的比较排序算法。许多编程语言的函数库中都提供了排序算法的接口，它们所使用的算法一般不是单一的排序算法，而是对多种排序算法进行结合。这种算法结合不会改进算法的理论性质，但更适合实际使用。

内省排序（Introspective Sort，Introsort）就是一种快速排序、插入排序和堆排序的混合排序算法，这一算法最早由 [Mus97] 提出。它的主体框架是快速排序，并使用递归实现，在两种情况下会切换到其他排序算法。

- 当递归达到一定层数的时候，改为堆排序，以避免递归层数太深而影响效率。
- 当待排序的子序列长度非常小时，采用插入排序。因为此时应用高级的排序算法反而会产生多余的开销。

这样一个简单的算法结合策略就使得内省排序超越了所有的经典排序算法。目前在 SGI C++、GNU C++、Microsoft .NET 的标准库中都采用内省排序来实现排序算法。

1.1.6 Timsort

Timsort 是由 Tim Peters 在 2002 年提出的排序算法，目前已经成为 Python 2.3 以后的版本内置的排序算法。Java SE 7、Android 平台、GNU Octave 也引入了这一排序算法。能够被多种编译器、解释器、工具采用，足以说明 Timsort 的实用性。

Timsort 是结合了归并排序和插入排序的混合排序算法。虽然插入排序的时间复杂度要高于归并排序，但当待排序序列非常短的时候，比较时间复杂度的意义是不大的。Timsort 的大致思想就是先采用插入排序将非常短的小段扩充为较长的小段，再采用归并排序来合并多个较长的小段。具体来说，我们需要定义一个参数 minrun，当小段长度小于 minrun 时，我们认为它是非常短的小段，否则认为它是较长的小段。下面我们分别描述 Timsort 是如何完成扩充和归并这两个步骤的。

我们从左至右处理待排序序列，将其划分成若干个小段。我们从第 1 个尚未处理的对象开始，找到一个尽可能长的连续严格递减或连续非递减序列，如果是连续严格递减序列，则可以通过一个简单的“翻转操作”在线性时间内将其变为严格递增序列。如果这样得到的序列长度不小于 minrun，则我们将其作为一个完整的小段，继续生成下一个小段；否则我们用插入排序将后面的元素添加进来，直至其长度达到 minrun 为止。我们考虑两个简单的例子。

- 待排序序列的前 4 个数是 $3,6,7,5$，$\text{minrun} = 4$，则尽可能长的连续非递减序列是 $3,6,7$，其长度没有达到 4。我们将 5 插入进来，得到长度为 4 的小段 $3,5,6,7$。
- 待排序序列前 4 个数是 $9,1,2,7$，$\text{minrun} = 4$，则尽可能长的连续递减序列是 $9,1$，翻转后为 $1,9$，其长度没有达到 4。我们依次将 2 和 7 插入进来，得到长度为 4 的小段 $1,2,7,9$。

我们再来考虑如何将小段合并。在理想情况下，我们应当尽量合并长度相近的小段，这样可以节约计算时间。使用霍夫曼树的归并策略虽然可行，但我们不应该花费太多时间在选择优先合并的小段上。Timsort 选择了一种折中的方案，它要求最右边的三个小段的长度尽量满足两个条件。我们记最右边的三个小段的长度从左至右分别是 A、B、C，则 Timsort 要求：

- $A > B + C$；
- $B > C$。

这样做的目的是让合并后的小段的长度从右至左以指数量级递增，这样我们只需从右至左依次进行合并就可以使每次合并的两个小段的长度大致相同。在具体实现上，如果

$A \leqslant B+C$，则我们合并 A、B 或者 B、C，这取决于哪一种合并方式生成的新小段更短。如果 $A > B+C$ 但 $B \leqslant C$，则我们合并 B、C。这一合并策略可以写为算法 8。

算法 8 合并小段

```
1: while 还有至少两个小段 do
2:   最右边的三个小段从左至右分别是 A、B、C（这里 A 不一定存在）。
3:   if A 存在且 A ⩽ B + C then
4:     if A < C then
5:       合并 AB。
6:     else
7:       合并 BC。
8:     end if
9:   else if B ⩽ C then
10:    合并 BC。
11:  else
12:    跳出循环。
13:  end if
14: end while
```

我们可以每生成一个新的小段都试图进行合并。在算法结束后，有可能会出现有剩余小段没有合并的情况。这时我们采用算法9进行强制合并，直到最终仅剩余一个小段，即排序结果。

算法 9 强制合并小段

```
1: while 还有至少两个小段 do
2:   最右边的三个小段从左至右分别是 A、B、C（这里 A 不一定存在）。
3:   if A 存在且 A < C then
4:     合并 AB。
5:   else
6:     合并 BC。
7:   end if
8: end while
```

我们来看一个具体的例子，考虑待排序序列

$$3,6,7,5,3,5,6,2,9,1,2,7,0,9,3,6,0,6,2,6,1,8$$

及 minrun $= 4$，则排序步骤如图 1.3所示。其中每一行代表 Timsort 的一个步骤。方块下的弯虚线表示在最初生成小段时首先找到的尽可能长的连续严格递减或连续非递减序列，弯实线表示小段。

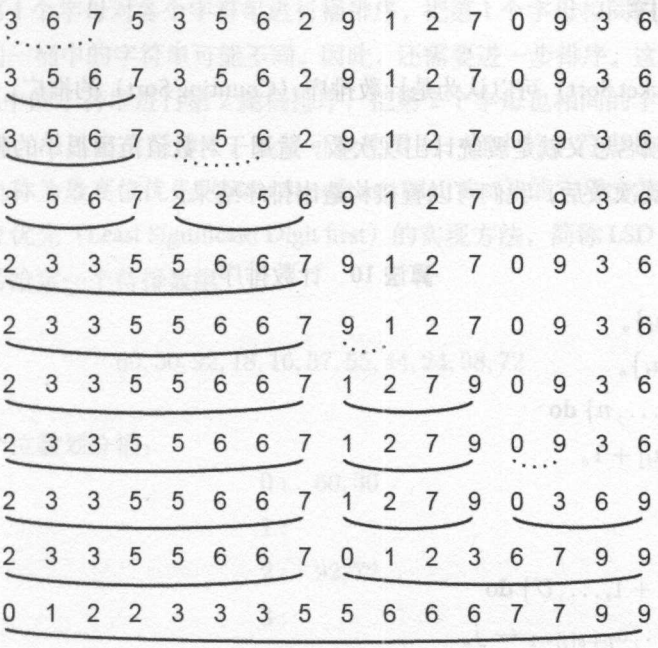


图 1.3 Timsort 示例

最后，我们还要讨论一下 minrun 的选取方式，虽然并不存在一个绝对正确的做法。我们假设待排序序列的初始顺序是比较随机的，则最初生成的小段长度基本上会是 minrun。因此，如果待排序序列的长度是 n，则我们总共会生成 $\left\lceil \frac{n}{\text{minrun}} \right\rceil$ 个初始小段。如果 $\left\lceil \frac{n}{\text{minrun}} \right\rceil$ 刚好是 2 的整数次幂，则归并过程将会非常“完美”，可以表述为一个满二叉树。如果 $\left\lceil \frac{n}{\text{minrun}} \right\rceil$ 比 2 的某个整数次幂稍大一点点，则到算法最后的阶段会出现一个超长小段与一个超短小段的合并，这是一种非常不好的情况。因此，我们会选取 minrun，使得 $\left\lceil \frac{n}{\text{minrun}} \right\rceil$ 刚好是 2 的整数次幂或比某个 2 的整数次幂稍小一点的数。

1.2 非比较排序

前面介绍的都是基于比较的排序，它们具有普适性但无法达到低于 $O(n\log n)$ 的时间复杂度。本节我们介绍一些非比较排序，它们的应用范围会小一些，但通常具有线性时间复杂度。

1.2.1 桶排序

桶排序（Bucket Sort）可以认为是计数排序（Counting Sort）的推广。

计数排序，顾名思义就是要统计出现次数，适用于对数值范围很小的整数序列排序。在统计每个数的出现次数后，我们可以直接构造出排序结果。

算法 10 计数排序

$L \leftarrow \min\limits_{1\leqslant i\leqslant n}\{a_i\}$。
$U \leftarrow \max\limits_{1\leqslant i\leqslant n}\{a_i\}$。
for $i=\{1,2,\ldots,n\}$ **do**
  $s[a_i] \leftarrow s[a_i]+1$。
end for
$p \leftarrow 1$。
for $j=\{L,L+1,\ldots,U\}$ **do**
  $a_p,a_{p+1},\ldots,a_{p+s[j]-1} \leftarrow j$。
  $p \leftarrow p+s[j]$。
end for

计数排序虽然简单，但应用面相对较窄，排序对象只能是整数，而且数值范围必须非常小。这些限制导致了计数排序的用途不是太大。但值得注意的是，这种把相同数字放到一起的思想和其他几种非比较排序在本质上是很类似的。

与计数排序相比，桶排序可以对更一般的对象进行排序。桶排序首先建立若干个按顺序排好的“桶”，然后将待排序的对象放入各自对应的桶中。由于不同桶之间的顺序已经确定，因此，当将所有对象都放入桶中之后就基本确定了排序结果。一般来说，同一桶中的对象是大小相同或相近的。如果大小相同则排序结束；如果大小相近则每个桶内可以再进一步排序。

注意到，桶的定义是很宽泛的。桶排序的效果也很依赖于桶的定义。如果待排序对象是整数，且每个整数都各自拥有一个独立的桶，则桶排序退化为计数排序。

1.2.2 基数排序

基数排序可以看作多次使用桶排序，特别适合按照字典序排序。基数排序将待排序对象划分为多个片段，逐一考虑各个片段，并针对每一片段进行一次桶排序。对于字符串而言，就是先根据第 1 个字母对各个字符串进行桶排序，把第 1 个字母相同的字符串全部放入同一桶中。由于同一桶中的字符串可能不同，因此，还需要进一步排序。这时就可以根据第 2 个字母对同一桶中的字符串进行第 2 轮桶排序，把第 2 个字母也相同的字符串全部放入同一桶中。以此类推，经过多次桶排序，最终可以使得同一桶中的字符串是完全相同的，从而排序结束。这也称为最高位优先（Most Significant Digit first）的实现方法，简称 MSD 法。与之对应的最低位优先（Least Significant Digit first）的实现方法，简称 LSD 法。举个整数排序的例子来说，若给定一个待排数组

$$60, 30, 92, 18, 16, 37, 55, 44, 24, 98, 72$$

则首先我们按照个位数划分桶：

$$\begin{aligned} 0&: \ 60, 30 \\ 1&: \\ 2&: \ 92, 72 \\ 3&: \\ 4&: \ 44, 24 \\ 5&: \ 55 \\ 6&: \ 16 \\ 7&: \ 37 \\ 8&: \ 18, 98 \\ 9&: \end{aligned} \tag{1.3}$$

连成一串就是：

$$60, 30, 92, 72, 44, 24, 55, 16, 37, 18, 98$$

然后保持这个先后次序按照十位数划分桶：

$$
\begin{aligned}
0&: \\
1&: \ 16, 18 \\
2&: \ 24 \\
3&: \ 30, 37 \\
4&: \ 44 \\
5&: \ 55 \\
6&: \ 60 \\
7&: \ 72 \\
8&: \\
9&: \ 92, 98
\end{aligned}
\tag{1.4}
$$

这样我们得到排序后的数组为：

$$16, 18, 24, 30, 37, 44, 55, 60, 72, 92, 98.$$

由于最多只到十位数，所以排序终止，否则可以再按照百位甚至更高位进行类似的操作。

1.3 总结

排序算法的种类繁多。通常我们不必知道排序算法的细节而只需直接调用库函数。即便如此，仍然有一些问题值得我们注意。

- 如果待排序对象是某种自定义数据结构，则我们通常不直接对这些数据进行排序，而是对它们的编号或索引进行排序。这是因为直接排序会导致数据结构的整体交换。如果数据结构较大，则会在交换上花费很多时间。但也有例外的情况存在，比如当内存占用更为关键的时候，直接排序会更好，这样不会占用很多额外的存储空间。
- 当遇到多个有序列合并、前 k 小数等特殊的排序问题时，有一种“万能”的方法是仍然直接调用库函数进行排序。但如果对计算时间要求较高，就需要我们针对特殊问题设计特殊的算法，这也就要求我们对排序算法必须有深入的理解。

第2章

哈希

2.1 基本概念与实现

哈希（Hash），也叫作散列、数据摘要等，是一类非常基础也非常实用的算法。哈希的核心概念主要包括哈希函数和哈希表。

2.1.1 哈希函数

哈希函数的作用就是把某一类不定长的对象映射为另一类固定长度的对象。能够做到这一点的函数可能有很多，是不是它们都适合作为哈希函数？如何比较哪种哈希函数才是好的哈希函数？这就是本节要讨论的问题。

如果两个不同的对象经过哈希函数计算后得到相同的哈希值，则这就是所谓的冲突（Collision）。冲突会导致许多问题，我们考虑一种极端情况：如果一个哈希函数的计算结果经常是0，那么它根本无法帮助我们区分不同的对象，也就不可能帮助我们快速查找了。因此，一个好的哈希函数应当尽量将不同的对象对应到不同的哈希值。此外，如果哈希函数比较简单，则有利于快速计算哈希值。

在设计哈希函数或者选择哈希函数的时候，我们主要考虑以下两个方面。

- 冲突少。即很少出现不同对象对应到相同哈希值的情况。
- 计算快。即计算哈希值能够快速完成。

对于一个具体的场景来说，哈希函数的设计可能需要我们非常精心和巧妙地运用先验知识。

从简单情况出发，我们首先考虑字符串哈希函数。这里的字符串特指 ASCII 编码的字符串，其字符集包括英文大小写字母、数字、标点符号等。一个很常用又很有效的算法是 djb2，其算法流程如下所示。

算法 11 djb2 算法

输入： 字符串 s。

输出： 哈希值 h。

1: $h \leftarrow 0$

2: **for** s 的每一个字符 c **do**

3: $\quad h \leftarrow h \times 2^6 + h + c$。

4: **end for**

这里乘以 2^6 在许多编程语言中都可以通过位运算来实现，因此速度很快。h 是一个无符号整型数，通常为 32 位或 64 位。我们不用关心 h 的溢出问题，因为溢出也可以视为一种取模。

与 djb2 算法很类似的还有 sdbm 算法。

算法 12 sdbm 算法

输入： 字符串 s。

输出： 哈希值 h。

1: $h \leftarrow 0$

2: **for** s 的每一个字符 c **do**

3: $\quad h \leftarrow h \times 2^6 + h \times 2^{16} - h + c$。

4: **end for**

虽然计算速度都很快，但对于哈希函数的冲突少这个特点，djb2 算法与 sdbm 算法都没有理论上的保证，而主要依赖于实验效果。在构造一个哈希函数后，往往需要采用大量的样本进行测试，根据冲突数量进行评估。

还有一类特殊的哈希函数，叫作密码学哈希函数（Cryptographic Hash Function）。这类哈希函数对冲突少的要求更高，它还要求很难构造冲突的例子。例如，$f(x) = x \mod 2^{64}$ 可能是一个不错的哈希函数，却不是一个好的密码学哈希函数，因为我们很容易构造冲突的例子，例如 $f(0) = f(2^{64}) = 0$。后面我们还会具体介绍密码学哈希函数与信息安全的联系。

2.1.2 哈希表

哈希表是一种数据结构，它需要配合哈希函数使用，用于建立索引，便于快速查找。

哈希表的实现一般来说就是一个定长的存储空间，每个位置存储一个对象。如果我们假设定长为 N，则可以将这 N 个存储位置分别编号为 $0, 1, \cdots, N-1$。那么现在的问题就是，如何决定一个元素应该放到哪个位置？如何查找一个元素在哪个位置？最常采用的策略如下。

- 插入：使用哈希函数计算待插入对象的哈希值，如果哈希值是 H，则插入编号为 $H \bmod N$ 的位置。
- 查找：使用哈希函数计算待查找对象的哈希值，如果哈希值是 H，则检查编号为 $H \bmod N$ 的位置。

我们通过一个例子来说明如何使用哈希表。假设我们有 4 个字符串需要建立索引，这 4 个字符串及其对应的哈希值如表 2.1所示。

表 2.1 字符串及其哈希值示例

字 符 串	哈 希 值
algorithm	1
mathematics	5
notes	9
book	15

如果哈希表的长度为 10，那么我们通常将其描述为图 2.1。其中，mathematics 和 book 两个字符串的哈希值模 10 的余数相同，因此放入同一位置。图中所采用的实际上是所谓的链地址法，即用链表将冲突的多个对象存储在同一位置，这也是在哈希表中解决冲突时最常用的方法。

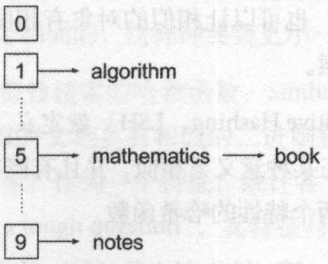


图 2.1 哈希表示例

当我们需要查找某个字符串是否在哈希表中时，只需计算其哈希值，然后到哈希表的对应位置检查，从而达到快速查找的目的。很容易看出，如果每个字符串都放在哈希表中的不同位置，则查找速度会更快。哈希函数冲突少的特点使得不同的字符串通常具有不同的哈希值，所以，一般来说，只要增大哈希表的长度，就可以避免将两个字符串放入同一位置的情况。

确定哈希表的长度是一个重要的难题。一方面，长度太小可能会导致冲突多，查询速度慢；另一方面，长度太长可能会白白浪费存储空间。如果事先知道需要索引的对象数目，则可以设定比较合适的哈希表大小，但许多时候并不能事先得到这个信息。一个实用的技巧是动态调整哈希表的长度：在开始时，首先选用一个较小的哈希表，当发现其使用率，即存储的对象数目与哈希表大小的比值达到某个阈值时，就建立一个更大的哈希表并将之前的哈希表中的对象迁移过来。迁移哈希表的过程往往是非常费时的，因此，可以同时保留新旧两个哈希表，逐步迁移，分散计算量。在进行查找和删除操作时，同时检查新旧两个哈希表。在进行插入操作时，只针对新的哈希表。每进行一次插入操作时，我们同时将 r 个对象从旧哈希表迁移到新哈希表，其中，r 是一个任意指定的常数。这个策略相当于每次从旧哈希表中删除 r 个对象，就同时在新哈希表中增加 $r+1$ 个对象。因此，只要新哈希表的长度不小于旧哈希表长度的 $\frac{r+1}{r}$ 倍，我们就可以在完全删除旧哈希表的时候，保证新哈希表的使用率没有超过旧哈希表。

2.2 哈希的应用

2.2.1 相似性搜索

哈希主要用于快速查找，而查找的对象需要是完全相同的。也就是说，一般只要求当两个对象完全相同时有相同的哈希值，而两个相似的对象的哈希值不需要有任何关系。但如果哈希函数设计得足够巧妙，也可以让相似的对象有相同或相似的哈希值，这样我们就可以借助哈希来进行相似性搜索。

局部敏感哈希（Local-Sensitive Hashing，LSH）就定义了一类可以衡量相似性的哈希函数，它要求相似对象的哈希值在某种意义下相似，并且有概率保证。用数学语言来描述，局部敏感哈希的定义为满足以下两个特性的哈希函数。

- 如果 $d(x,y) \leqslant d_1$，则 $P(\text{hash}(x) = \text{hash}(y)) \geqslant p_1$。也就是说，当两个对象的距离不大于 d_1 时，它们的哈希值相同的概率要不小于 p_1。

- 如果 $d(x,y) \geqslant d_2$，则 $P(\text{hash}(x) = \text{hash}(y)) \leqslant p_2$。也就是说，当两个对象的距离不小于 d_2 时，它们的哈希值相同的概率要不大于 p_2。

其中，$d(x,y)$ 表示两个对象 x 和 y 之间的距离，$\text{hash}(\cdot)$ 是哈希函数，$P(\cdot)$ 是概率，d_1、d_2、p_1、p_2 都是常数。对于不同的场景，对距离的定义可能是不同的，因此，哈希函数的设计也会很不一样。

我们来看一个局部敏感哈希的具体例子，在这个例子中考虑两个集合之间的距离。

定义 2.1（Jaccard 距离） 两个集合 $\mathcal{A}, \mathcal{B}$ 之间的距离定义为 $1 - \dfrac{|\mathcal{A} \cap \mathcal{B}|}{|\mathcal{A} \cup \mathcal{B}|}$。

根据定义，我们可以验证：

- 如果两个集合完全相同，则它们的 Jaccard 距离应当等于 0；
- 如果两个集合完全不同，则它们的 Jaccard 距离应当等于 1；
- 两个集合的 Jaccard 距离就是它们的不同元素占所有元素的比例。

针对这样的距离定义，最小哈希（Minhash）就是一种局部敏感哈希。计算最小哈希需要事先对元素的总集合随机排序，得到一个排列 $\mathcal{O}$。对于任意给定的集合，其哈希值就是它的各个元素在 $\mathcal{O}$ 中排名最靠前的一个。例如元素总集合为 $\{a,b,c,d,e,f\}$，我们对其进行随机排序，得到 e,f,b,c,a,d，则集合 a,b,c 的哈希值为 b，集合 c,d 的哈希值为 c，集合 b,e,f 的哈希值为 e，集合 a,e,f 的哈希值为 e。很容易计算出两个集合 $\mathcal{A}$、$\mathcal{B}$ 的哈希值相同的概率为 $\dfrac{|\mathcal{A} \cap \mathcal{B}|}{|\mathcal{A} \cup \mathcal{B}|}$，这是因为，它们的并集中的各个元素在 $\mathcal{O}$ 中排名最靠前的一个需要是它们交集中的元素。与 Jaccard 距离的定义对比可以发现，最小哈希满足局部敏感哈希的定义，即：

- 如果 $d(\mathcal{A},\mathcal{B}) \leqslant d_1$，则 $P(\text{hash}(\mathcal{A}) = \text{hash}(\mathcal{B})) \geqslant 1 - d_1$；
- 如果 $d(\mathcal{A},\mathcal{B}) \geqslant d_2$，则 $P(\text{hash}(\mathcal{A}) = \text{hash}(\mathcal{B})) \leqslant 1 - d_2$。

其中，d_1 和 d_2 可以在 $[0,1]$ 范围内任意选取。由于不同的排列可以定义不同的最小哈希，所以在实际使用最小哈希的时候，还可以定义多个哈希函数。当两个集合的多个哈希值都相同的时候，才认为它们的哈希值是相同的，这样冲突会更小一些。

下面再来看一个用于文档相似性搜索的哈希函数：Simhash。Simhash 可以将文档哈希到一个 64 位二进制数，使得相似的文档具有相似的二进制数。对于一个文档，我们可以把文中的每个词（也可以是词组等）作为一个特征，统计各个特征的出现频率。例如，我们考虑文档“To be or not to be is a tough question”，其特征与相应的频率为：(to, 0.2)、(be, 0.2)、(or, 0.1)、(not, 0.1)、(is, 0.1)、(a, 0.1)、(tough, 0.1)、(question, 0.1)。然后，我们可以利用某种传统的哈希函数将特征映射到 64 位二进制数。这里为了描述方便，我们只映射到

6 位二进制数，并假设映射结果是 to: 001100, be: 100100, or: 001010, not:101000, is: 011100, a: 100101, tough: 100111, question: 011011。根据二进制数的各个二进制位，我们对每个特征构造一个向量。如果一个特征映射到的二进制数的某一位是 1，则其向量对应位置上的分量为该特征的频率，否则为频率的相反数。于是，各个特征对应的向量为

$$
\begin{aligned}
\text{to:} &\quad (-0.2, -0.2, 0.2, 0.2, -0.2, -0.2),\\
\text{be:} &\quad (0.2, -0.2, -0.2, 0.2, -0.2, -0.2),\\
\text{or:} &\quad (-0.1, -0.1, 0.1, -0.1, 0.1, -0.1),\\
\text{not:} &\quad (0.1, -0.1, 0.1, -0.1, -0.1, -0.1),\\
\text{is:} &\quad (-0.1, 0.1, 0.1, 0.1, -0.1, -0.1),\\
\text{a:} &\quad (0.1, -0.1, -0.1, 0.1, -0.1, 0.1),\\
\text{tough:} &\quad (0.1, -0.1, -0.1, 0.1, 0.1, 0.1),\\
\text{question:} &\quad (-0.1, 0.1, 0.1, -0.1, 0.1, 0.1).
\end{aligned}
\tag{2.1}
$$

将这些向量相加，就得到 $(0, -0.6, 0.2, 0.6, -0.2, -0.4)$。对于这一向量的每个分量，如果大于 0 就取 1，否则就取 0，这样得到的二进制数就是 Simhash 的最终哈希值，即 001100。可以想象，出现频率高的特征，其对应的向量分量的绝对值更大，对最终向量相加的结果的影响也更大。因此，如果两个文档相似，那么它们出现频率高的特征应该比较接近，最终得到的哈希值也就会有很多二进制位都是相同的。在查找相似文档的时候，我们只需要找到哈希值的各个二进制位区别很小的文档。一般来说，我们要求至多有 3 个二进制位不同。

还有一类主要用于图片相似性搜索的哈希函数是感知哈希（Perceptual Hashing）。它是一类广泛使用的图片搜索算法，可以做到以图搜图。其思想主要是通过调整大小、灰度化和二值化三个步骤，将图片映射到一个二进制数，使得相似图片具有相近的二进制数。我们以图像处理中的经典图片 Lenna 为例进行说明。

Lenna 原图如图 2.2所示。假设我们希望哈希值是 64 位二进制数，则我们首先将图片大小通过简单的缩放调整为 8像素 × 8像素，这样我们就刚好有 64 个像素了。在调整图片大小得到的新图中，每个像素上的 RGB 值实质上是原图相应位置上的 RGB 平均值。然后，我们将原图灰度化。灰度化最常见的方式就是将每个像素上的灰度值取为 RGB 三个分量的平均值。

这时我们得到图 2.3。尽管已经完全看不出原图，但在色彩明暗上，它仍然保留了原图的一些特征。

图 2.2　Lenna 原图

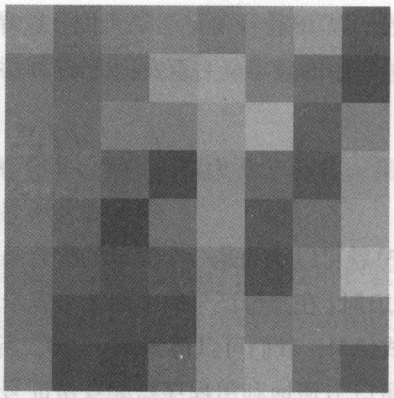


图 2.3　调整大小及灰度化后的 Lenna 图

图 2.3中各个像素点的灰度值可以写为矩阵

$$\begin{pmatrix} 88 & 51 & 68 & 75 & 62 & 78 & 102 & 31 \\ 64 & 49 & 56 & 107 & 125 & 72 & 54 & 23 \\ 63 & 53 & 72 & 71 & 107 & 136 & 41 & 74 \\ 65 & 57 & 45 & 22 & 99 & 52 & 33 & 98 \\ 61 & 46 & 17 & 55 & 100 & 35 & 58 & 109 \\ 56 & 38 & 27 & 35 & 81 & 27 & 76 & 147 \\ 59 & 26 & 24 & 24 & 93 & 67 & 76 & 72 \\ 66 & 20 & 17 & 49 & 95 & 116 & 57 & 36 \end{pmatrix} \tag{2.2}$$

最后一步是二值化。我们首先计算出 64 个像素点的灰度平均值为 63.406，然后将小于平均值的像素点都取为 0，其余像素点都取为 1。将所有 64 个 0 或 1 连在一起，就得到了一个 64 位二进制串，也就是最终的哈希值：

1011011010011100001111011000100100001001000010110000111110001100

这是我们只需对两个图片的哈希值进行逐位比较，就可以判定它们是否相似了。

2.2.2　信息安全

哈希在信息安全中主要用于数据校验：通过比对哈希值来验证数据的正确性和完整性。我们在现实生活中就经常遇到这样的例子，包括身份证号码和信用卡号码。

我们以18位身份证号码为例，这样读者也可以根据自己的身份证号码验证。我们将身份证号码中的数字从右至左记为 $a_1, a_2, \cdots, a_{18}$。其中最右边一位 a_1 是校验位，满足

$$a_1 = \left(12 - \left(\left(\sum_{i=2}^{18} a_i 2^{i-1}\right) \mod 11\right)\right) \mod 11 \tag{2.3}$$

式(2.3)就可以认为是一个简单的哈希函数。哈希值 a_1 有可能是 $0-10$ 中的任意一个数字。当 $a_1 = 10$ 时，在身份证号码中用X来表示。例如，身份证号码的前17位是11010519491231002，根据式(2.3)，我们可以得到 $a_1 = 10$，所以最后一位必须是X，也就是10。这样的设计使得我们可以方便地利用程序检测身份证号码是否输入正确。可能有人会好奇，为什么我们要模11？如果选择模10，就不必出现X了。因为这样的选择使得我们在写错身份证号码中任何一个数字的时候，都导致公式不成立。假设我们把某个 $a_k (k \in \{2, 3, \cdots, 18\})$ 误写为 $a_k + t$，其中 $-9 \leqslant t \leqslant 9$，则根据式(2.3)，$a_1$ 会变为 $(a_1 + 2^{k-1}t) \mod 11$。如果我们希望 a_1 保持不变，则必须有 $(2^{k-1}t) \mod 11 = 0$，但这个等式在当且仅当 $t = 0$ 时成立。也就是说只要写错一位，就必然导致校验码 a_1 发生变化。换句话说，任何两个合法的身份证号码至少会有两位数字是不同的。读者可以验证，模10没有类似的性质。

信用卡号码也有类似的校验规则，我们经常使用的是Luhn算法。Luhn算法是以IBM科学家Hans Peter Luhn的名字命名的。这一算法与身份证校验的算法类似。由于信用卡号码的长度不统一，所以我们记 n 为信用卡号码的长度，号码中的数字从右至左记为 $a_1, a_2, \cdots, a_n$。其中 a_1 是校验位。我们定义

$$b_i = (2 - (i \mod 2))\, a_i, \forall i = 2, 3, \cdots, n \tag{2.4}$$

则

$$a_1 = \left(10 - \left(\left(\sum_{i=2}^{n} (b_i \mod 10) + \left\lfloor \frac{b_i}{10} \right\rfloor\right) \mod 10\right)\right) \mod 10 \tag{2.5}$$

例如，信用卡号除最右边一位外是4992739871，则按照公式计算，我们可以得到最后一位应当是6。与身份证号码类似，任何一位信用卡号码的变化，也会导致校验位的变化。我们可以根据发生变化的一位是奇数位还是偶数位分别给出证明。具体过程留给感兴趣的读者。

另一个常见的校验是文件完整性校验，也就是说检查文件是否完整。什么时候文件会不完整？我们举两个例子来说明。

- 文件被篡改。比如我们需要在某个软件官方网站下载应用，但是文件很大，官方网站传输速度不理想。于是我们更换到了另外一个网站下载该应用，但不能确定是否同官

方网站的文件一致。官方网站通常会给出一个校验码，即哈希值。我们可以验证该校验码是否与所下载的文件一致，如果不一致则说明我们下载到的文件不是原版，有可能被篡改过。

- 文件下载不完整。在下载文件过程中，由于传输错误有可能导致文件不完整，我们同样可以对比官方网站给出的校验码，检查文件是否完整。

身份证号码和信用卡号码的校验主要是为了避免输入错误，但文件完整性的校验必须有防止伪造的功能。专门用于防止伪造这类特殊需求的哈希函数就是密码学哈希函数。理想的密码学哈希函数具有以下特殊性质。

- 很难生成一个数据，使其具有给定的哈希值。
- 很难找到两个具有相同哈希值的不同数据。

这两点特殊性质也可以用于衡量密码学哈希函数的安全性。在理想情况下，除枚举外我们没有更好的办法生成一个数据，使其具有给定的哈希值。所谓的破解某个密码学哈希函数，就是指找到比枚举更快的方法。常见的密码学哈希函数包括 MD5、SHA-0、SHA-1、SHA-2、SHA-3 等。为了有足够高的安全性，这些密码学哈希函数的加密过程都比较复杂。然而所谓安全性，一般来说都不会有数学证明，也就难免会被人破解。即使没有被破解，随着计算机硬件的飞速发展，曾经安全的密码学哈希函数也会逐渐变得不安全。一般来说，较早提出的算法的安全性更差，会逐渐被时代淘汰，因此在安全性要求较高的情况下，需要选用业界认可尚未过时的密码学哈希函数。

密码学哈希函数在密码存储方面也有很大的作用。早期的网站存储密码的方式都是明文存储的。一旦发生数据泄漏，其他人就可以直接看到密码。而现在一般是使用某个密码学哈希函数加密，只存储其哈希值。在验证密码正确性的时候，我们只需要计算哈希值是否与记录一致。目前几乎所有大型网站在保存用户密码的时候都采用了密码学哈希函数。一方面，在验证用户密码的时候只需要计算密码的哈希值再进行比对，过程非常简单；另一方面，即使网站因此有数据泄漏，除了枚举也没有更好的方式可以知道密码。

2.2.3 比特币

可能大部分人对比特币的了解只停留在它是一种电子货币。如果对算法设计很感兴趣，那么很有必要了解一下比特币的原理，因为其中包含了很多颇具技巧性的算法设计。这里只涉及比特币的挖矿部分。

获得比特币的主要途径有两种，一种是交易，另一种就是挖矿。比特币的设计者采用了挖矿机制来产生新的比特币，以这种方式模拟货币的增发。挖矿是一个很形象的描述，实际上是要计算一类问题。粗略地来说，这类问题可以这样描述。

定义 2.2（比特币的挖矿问题的简化版） 给定一个字符串 S，需要找出一个字符串 N，使得 $S+N$ 即两个字符串连接后得到的新字符串在两轮 SHA-256 哈希后得到的数小于一个目标值 T。

由于 SHA-256 是密码学哈希函数，我们可以认为得到的哈希值是非常随机的，因此要解决挖矿问题，只能枚举字符串 N。对于随机尝试的一个 N，挖矿成功的概率约为 $T/2^{256}$，仅与目标值 T 有关，而与字符串 S 和 N 无关。如果源源不断地尝试 N，则期望是在 $2^{256}/T$ 次尝试后挖矿成功。因此比特币的设计者通过调整 T 来控制挖矿的难度，T 越大挖矿难度越小。在不知道全世界有多少计算机在挖矿的情况下，比特币的设计者采用了动态调整 T 的机制，使得平均每 600 秒会有人挖矿成功一次。假设前一次挖矿的目标值是 T_{prev}，挖矿成功花费的时间是 M 秒，那么我们可以推算出全世界每秒尝试 N 的次数大约是 $2^{256}/(T_{\text{prev}}M)$。如果后面依然保持这个计算速度，为了在大约花费 600 秒时刚好解决下一个挖矿问题，则可以算出下一个目标值 T 应当设定为 $T_{\text{prev}}M/600$。这种调整策略可以维持比特币每 600 秒增发一次，这样就控制了增发的速度。回顾整个过程可以发现，这个设计的正确性强烈依赖于密码学哈希函数的特点。一旦有人破解了 SHA-256，即发现比枚举更快的算法解决挖矿问题，那么比特币的整个设计都会出现问题。

2.2.4 负载均衡

我们在使用多台服务器处理请求时，一个很基本的问题是如何分配请求。假设我们有 N 台服务器，编号分别为 $0,1,2,\cdots,N-1$，那么我们可以计算请求的哈希值，然后对 N 取模。如果得到的结果是 i，那么我们就把这个请求发到编号为 i 的服务器。一个好的哈希函数可以使请求均匀地分配到各台服务器上，保证负载均衡。

但在实际操作中，服务器的数量可能是变化的。例如，任务多的时候我们可能会增加服务器的数量，任务少的时候我们可能会减少服务器的数量。即使不主动调整服务器的数量，在有服务器出现故障时，我们也不得不被动地减少服务器的数量。如果我们使用上述哈希方法，则由于服务器数量 N 的变化，大多数请求都会更换服务器。这可能会引起大量的数据迁移，容易给后端数据库和硬盘带来压力。而在理想情况下，将服务器的数量从 N 变为 $N-1$ 只会导致 $1/N$ 的请求更换服务器，将服务器数量从 N 变为 $N+1$ 只会导致 $1/(N+1)$ 的请求更换服务器。

一致性哈希（Consistent Hashing）可以用于解决上述难题。在服务器数量发生改变时，它可以使得数据迁移尽量小。其算法思路非常巧妙。我们选择两个合适的哈希函数，将请求和服务器分别映射到一个区间上的整数，例如 $[0, 2^{32}-1]$。这时我们可以把请求和服务器都标记在一个环上。每个请求都由其顺时针方向的第 1 个服务器来处理。

图 2.4 给出了一致性哈希的示例。我们将服务器和请求的哈希值标记在环上。其中 S1、S2、S3 代表服务器，Q1、Q2……Q6 代表请求。根据我们的规则，Q1、Q2、Q6 由 S1 来处理，Q3、Q4 由 S2 来处理，Q5 由 S3 来处理。

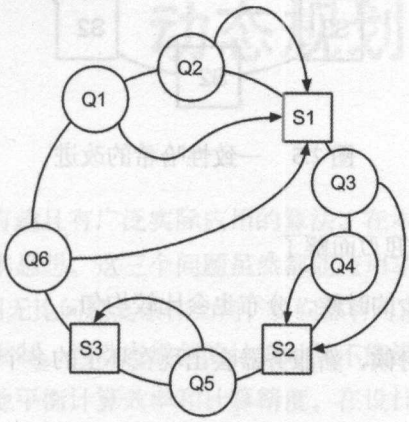


图 2.4 一致性哈希示例

如果增加一台服务器，则我们只需将一小部分请求给新的服务器处理；如果减少一个服务器，则我们只需将它本应处理的请求转移至下一个服务器。其余大部分请求都可以保持原来的逻辑不变。

但是这样做依然有一些弊端。

- 在服务器数量较少的时候，它们在环上的分布有可能非常不均匀，会有一些服务器承担很多的请求量，也会有一些服务器承担很少的请求量。
- 在增加一台服务器的时候，只能减轻某一台服务器的请求量，而其他服务器都不会有改善。
- 在减少一台服务器的时候，会有某一台服务器突然承受大约两倍的请求量。

这些弊端也是可以解决的。我们可以为服务器定义多个哈希函数，并将它们全部标记在环上。如图 2.5所示，三台服务器均被标记了三次。仍然按照之前的规则，每个请求都由其顺时针方向的第 1 个服务器来处理。

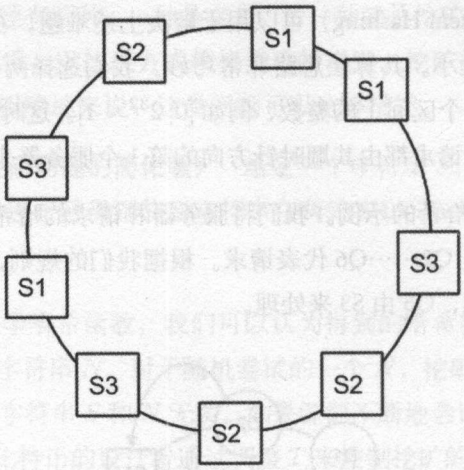


图 2.5　一致性哈希的改进

这时之前的弊端就全部迎刃而解了。

- 即使在服务器数量较少的时候，分布也会比较均匀。
- 在增加一台服务器的时候，新服务器会出现在环上的多个位置，因此能够减轻多台服务器的请求量。
- 在减少一台服务器的时候，请求量会比较均匀地分担给多台服务器。

第3章

动态规划与近似算法

动态规划是一类非常有趣且有广泛实际应用的算法。在本章中我们通过三个问题来帮助大家理解动态规划的基本思想。这三个问题虽然都能应用动态规划求解，但在计算复杂性上有本质的区别。这表明无论问题复杂性如何，我们都有可能运用动态规划求解，只是实际效果会受到问题规模的限制。当动态规划的计算时间不能满足要求时，我们可以考虑近似算法，这样就可以很好地平衡计算效率和计算精度。在设计近似算法时，我们也常常需要借鉴动态规划算法的思想。两类算法的结合，可以很好地处理不同规模的同一问题：动态规划直接处理小规模问题，再结合近似算法处理大规模问题。

3.1 基本概念

3.1.1 动态规划

在20世纪50年代初，美国数学家Richard E. Bellman等人在研究多阶段决策过程(Multistep Decision Process)的优化问题时，提出了著名的最优化原理(Principle of Optimality)，把多阶段过程转化为一系列单阶段子问题，利用各阶段之间的关系逐个求解，创立了解决这类过程优化问题的新方法——动态规划，并于1957年出版了该领域的第1本著作。

动态规划适用的问题都具有最优子结构的性质，即原问题最优可以归结为子问题最优，通常我们需要执行以下三个步骤。

(1) 定义子问题，即定义状态。

(2) 定义状态转移规则，可以理解为状态之间的递推关系。

(3) 定义初始状态。

按照这三个步骤，我们就可以使用动态规划求解原问题了。

3.1.2 计算复杂性

本章会提到 $\mathcal{P}$-问题、$\mathcal{NP}$-难问题、强 $\mathcal{NP}$-难问题，这些概念都与近似算法密切相关。

所谓 $\mathcal{P}$-问题，是指确定性图灵机可以在多项式时间内求解的问题，简单来说，就是可以在多项式时间内找到解。

所谓 $\mathcal{NP}$-问题，是指非确定性图灵机可以在多项式时间内求解的问题，就是可以在多项式时间内验证一个解是否正确。验证一个解是否正确当然要比找到解更容易，因此 $\mathcal{P} \subseteq \mathcal{NP}$。而 $\mathcal{P} = \mathcal{NP}$ 是否成立，至今仍是未解难题，但一般认为 $\mathcal{P} \neq \mathcal{NP}$，也就是说 $\mathcal{NP}$-问题包含一些不能在多项式时间内找到解的问题。$\mathcal{NP}$-难问题则是比 $\mathcal{NP}$-问题更难的一类问题，所有 $\mathcal{NP}$-问题可以在多项式时间内归约成 $\mathcal{NP}$-难问题。在 $\mathcal{NP}$-难问题中有一些存在伪多项式算法。所谓伪多项式算法就是指计算时间关于输入数值的大小是多项式，但关于输入数据的长度并不是多项式。强 $\mathcal{NP}$-难问题则不存在伪多项式算法。本章将要讨论的子集和问题属于 $\mathcal{NP}$-难问题，旅行商问题则属于强 $\mathcal{NP}$-难问题。

对于 $\mathcal{P}$-问题，我们主要研究多项式时间算法。而对于 $\mathcal{NP}$-难问题和强 $\mathcal{NP}$-难问题，则主要研究近似算法。假设问题的最优值是 z^*，近似算法求得的结果一般是非常接近 z^* 的。如果近似算法求得的结果总是在 αz^* 和 z^* 之间，则我们称近似比为 α。很明显，近似比 α 越接近 1，说明近似效果越好。

3.2 字符串的编辑距离

编辑距离，是指将一个对象编辑为另外一个对象的操作次数或代价。这里的对象很宽泛，对于每一类具体的对象需要更完善的定义。编辑距离可以用于衡量相似度，例如适当地定义图编辑距离后，我们就可以计算图像的相似度。本节我们考虑的是字符串编辑距离，因为字符串相对简单、易于理解，用途也很广泛。如无特殊说明，本节所讨论的编辑距离都特指字符串的编辑距离。

3.2.1 问题引入

首先，我们给出编辑距离的定义。

定义 3.1（编辑距离，Edit Distance） 我们要对一个字符串进行若干编辑操作，编辑操作包括插入一个字符、删除一个字符及替换一个字符。给定一个原始字符串 S_1 和一个目标字符串 S_2，我们将 S_1 到 S_2 的编辑距离定义为 S_1 修改成 S_2 所需的最小编辑次数。

举例来说，hat 到 what 的编辑距离是 1，因为将 hat 修改为 what 需要至少一次操作（插入 w）；view 到 new 的编辑距离是 2，因为将 view 修改为 new 需要至少两次操作（删除 v，将 i 替换为 n）。编辑方式可能不唯一，例如将 see 修改为 tree 可以先将 s 替换为 t，再插入 r，也可以先将 s 替换为 r，再插入 t。

事实上编辑距离还有多种不同的定义方式，在本节中我们考虑的仅仅是其中一种形式：Levenshtein 距离，它是以科学家 Vladimir Levenshtein 的名字命名的。我们通常所说的字符串编辑距离，一般特指 Levenshtein 距离，这也是最常见的字符串编辑距离。其他常见的字符串编辑距离还包括如下几种。

- Damerau Levenshtein 距离。除了插入、删除、修改这三种操作，它还考虑了交换两个相邻字母的操作。例如 dairy 到 diary 的 Damerau Levenshtein 距离是 1，而 Levenshtein 距离是 2。
- 最长公共子序列距离（Longest Common Subsequence Distance）。仅考虑插入、删除两种操作。
- 汉明距离（Hamming Distance）。仅考虑修改操作，因此两个字符串的长度需要一致。

此外我们还可以对不同的操作赋予不同的权重，甚至根据所操作的字母来确定权重，从而定义新的编辑距离。本节我们将要讨论的许多算法思想都可以类似地应用于其他编辑距离的变种，有兴趣的读者可以查阅资料或自行尝试。

编辑距离具有以下几个性质。

- 编辑距离是对称的，即 S_1 到 S_2 的编辑距离等于 S_2 到 S_1 的编辑距离。这本质上是因为插入和删除是互逆操作，替换的逆操作还是替换。因此，我们也可以称 S_1 到 S_2 的编辑距离为 S_1 和 S_2 的编辑距离。
- 两个字符串的编辑距离不会大于长字符串的长度。这一点很容易理解，比如将长字符串 S_1 修改为短字符串 S_2，我们可以对 S_1 开头的若干位置执行替换操作得到 S_2，再

删除结尾的多余字符。此时操作次数为长字符串的长度，编辑距离一定不会大于这个值。

- 两个字符串的编辑距离不会小于字符串长度之差。因为将长字符串 S_1 修改为短字符串 S_2，至少要执行删除多余字符的操作。
- 当且仅当完全相同时，两个字符串的编辑距离为 0。
- 如果两个字符串的长度相同，则它们的编辑距离不会大于汉明距离。因为这时只执行替换操作的话，编辑次数刚好等于汉明距离。
- 编辑距离满足三角不等式，即 S_1 和 S_2 的编辑距离一定不大于 S_1 和 S_3 的编辑距离与 S_3 和 S_2 的编辑距离之和。这是因为我们至少可以把 S_1 先修改为 S_3，再进一步修改为 S_2。

以上这些性质除了帮助我们理解编辑距离，还可以用于算法优化。

编辑距离常常用来衡量两个字符串的相似程度。编辑距离越小则越相似，这是非常直观的一种相似度衡量方式。编辑距离有以下两个重要应用。

- 拼写矫正：当用户在字典中搜索一个词的时候，很可能由于拼写错误导致没有查询结果。此时，如果我们可以为用户提供编辑距离最相近的一些词作为建议查询词，就可以很好地提升字典服务的用户体验。例如在某文字编辑软件中输入 ative 后，由于字典中不存在这一单词，该软件提示了 5 个修改建议，如图 3.1所示。我们可以看到前 4 个修改建议与输入词 ative 的编辑距离都是 1，最后一个修改建议与输入词的编辑距离是 2。尽管我们不清楚该软件实际所用的算法，但编辑距离排序看上去是非常合理的。对于键盘输入这种最普遍的输入方式，我们如果考虑两个相邻字母交换、字母连续敲击两次和键盘上字母的位置等因素，给予这些操作不同的权重，则纠错效果可能会更好。

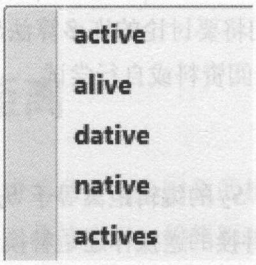


图 3.1 对输入词 ative 的拼写矫正

- DNA 分析：DNA 序列可以看作由 ATCG 组成的字符串，因此我们可以根据编辑距离去评估 DNA 的相似程度。与拼写矫正不同的是，我们有时还需要知道具体的编辑过程。有一个非常重要的概念叫作序列比对或序列对齐（Sequence Alignment），它实际上就是要将两个 DNA 序列并排列出，尽可能地标出相同的部分。例如

$$\begin{array}{lcccccccccccc} S_1: & T & C & A & G & G & T & C & T & - & A & G \\ & | & | & | & | & & | & & | & & | & | \\ S_2: & T & C & A & G & - & T & T & T & A & A & G \end{array} \tag{3.1}$$

这个示例中的横线表示对齐 S_1 与 S_2 后产生的间隔。将这个例子对应到字符串编辑，可以理解为通过一次删除（字母 G）、一次替换（字母 C 变为字母 T）和一次插入（字母 A）可以将 S_1 变为 S_2。

这里包含了两个重要问题：一个是如何计算编辑距离，另一个是如何计算编辑过程。前者可以用于判定序列的相似程度，后者可以用于序列对齐。我们在下面几节中详细地回答了这两个问题。

3.2.2 动态规划算法

我们记 S_1 的字符串长度为 n_1，S_2 的字符串长度为 n_2，记 $S(i)$ 表示字符串 S 的第 i 个字符，记 $d[i,j]$ 表示 S_1 的前 i 个字符所构成的前缀与 S_2 的前 j 个字符所构成的前缀之间的编辑距离。显然 $d[n_1,n_2]$ 就是我们最终要求的编辑距离。$d[i,j]$ 的定义是使用动态规划求解编辑距离问题时非常重要的一环，它将一个大问题转化成了若干个彼此接近的小问题。我们可以对末尾字符进行分类讨论，得出 $d[i,j]$ 的递推式。

首先，如果在两个前缀中有一个是空串，则编辑距离等于另一字符串的长度。

$$d[i,0]=i \tag{3.2}$$

这是因为原始字符串包含 i 个字符，而目标字符串为空，我们需要删除 i 个字符。根据编辑距离的对称性，我们得到

$$d[0,j]=j \tag{3.3}$$

这是因为原始字符串为空，而目标字符串包含 j 个字符。我们需要插入 j 个字符。

如果两个前缀都不是空串，则有以下递推式。

$$d[i,j]=\begin{cases} d[i-1,j-1], & S_1(i)=S_2(j) \\ \min\begin{cases} d[i,j-1]+1, \\ d[i-1,j]+1, \\ d[i-1,j-1]+1, \end{cases} & S_1(i)\neq S_2(j) \end{cases} \tag{3.4}$$

这个式子包含了几种情况：如果 $S_1(i)=S_2(j)$，那么只需要考虑 S_1 的前 $i-1$ 个字符与 S_2 的前 $j-1$ 个字符；如果 $S_1(i)\neq S_2(j)$，那么需要考虑对 S_1 的末尾进行插入、删除和替换三种情况。

下面我们来计算 S_1 ="math" 和 S_2 ="mouth" 的编辑距离。由于这两个单词很简单，所以我们可以直接看出它们的编辑距离是 2，具体编辑过程可以是：

(1) math → moth（将 a 替换为 o）；

(2) moth → mouth（插入 u）。

所有的 $d[i,j]$ 实质上构成了一个矩阵，如表 3.1所示。右下角的数字就代表 $d[4,5]=2$，也就是我们最终要求的编辑距离。实际上动态规划的过程就是填满这个表格的过程。

表 3.1 动态规划求编辑距离的具体过程

		m	o	u	t	h
	0	1	2	3	4	5
m	1	0	1	2	3	4
a	2	1	1	2	3	4
t	3	2	2	2	2	3
h	4	3	3	3	3	2

根据以上分析，我们可以实现一个时间复杂度和空间复杂度均为 $O(n_1n_2)$ 的算法。具体过程如算法13所示。

算法 13 编辑距离的动态规划算法

1: **for** $j\in\{0,1,\ldots,n_2\}$ **do**
2: $d[0,j]\leftarrow j$。
3: **end for**
4: **for** $i\in\{1,2,\ldots,n_1\}$ **do**
5: $d[i,0]\leftarrow i$。
6: **for** $j\in\{1,2,\ldots,n_2\}$ **do**

```
    if S_1(i) = S_2(j) then
      d[i, j] ← d[i − 1, j − 1]。
    else
      d[i, j] ← min{d[i, j − 1], d[i − 1, j], d[i − 1, j − 1]} + 1。
    end if
  end for
end for
```

算法13也被称为 Wagner Fischer 算法。这一算法在 20 世纪六七十年代被包括 [WF74] 在内的许多文章独立提出。

3.2.3 滚动数组优化

滚动数组优化是动态规划中的常见技巧。在编辑距离的动态规划算法中，在计算 $d[i,\cdot]$ 时，我们仅仅用到了 $d[i-1,\cdot]$，这就说明我们已经没有必要存储 $d[0,\cdot], d[1,\cdot], \cdots, d[i-2,\cdot]$。因此我们可以开辟两个长度为 n_2 的存储空间，分别用于存储 $d[i,\cdot]$ 和 $d[i-1,\cdot]$。在计算完 $d[i,\cdot]$ 之后，我们就可以丢弃 $d[i-1,\cdot]$，将那一部分空间用于存储 $d[i+1,\cdot]$，这就是所谓的滚动数组技巧。这一技巧不影响时间复杂度，但是空间复杂度可以降低至 $O(n_2)$。若 $n_1 < n_2$，则可以通过交换两个字符串使得空间复杂度变为 $O(n_1)$，因此空间复杂度可以降低至 $O(\min\{n_1, n_2\})$。

算法 14 编辑距离的动态规划算法（滚动数组优化）

```
for j ∈ {0, 1, ..., n_2} do
  d^old[j] ← j。
end for
for i ∈ {1, 2, ..., n_1} do
  d^new[0] ← i。
  for j ∈ {1, 2, ..., n_2} do
    if S_1(i) = S_2(j) then
      d^new[j] ← d^old[j − 1]。
    else
      d^new[j] ← min{d^new[j − 1], d^old[j], d^old[j − 1]} + 1。
    end if
  end for
```

13: 交换 d^{old} 和 d^{new}。

14: **end for**

注意算法14的第 13 行在程序实现时，并不必真的交换存储的数据，而只需交换存储空间的地址。例如在 C++ 中只需交换二者的指针。

3.2.4 上界限制

在实际问题中，编辑距离主要用于衡量字符串的相似度。通常只有在两个字符串比较相近的时候，我们才需要精确地计算它们的相似度，它们的编辑距离才有意义。如果编辑距离很大，就意味着两个字符串的差距很大，这时也就没有必要精确计算。针对这样一种情况，我们可以设定一个常数 D，仅当编辑距离小于 D 时才精确计算。一般来说 D 远小于 n_1 和 n_2。

我们曾经提到编辑距离具有这一性质：两个字符串的编辑距离不会小于字符串长度之差，即

$$d[i,j] \geqslant |i-j| \tag{3.5}$$

由此可知，当 $|i-j| \geqslant D$ 时，$d[i,j] \geqslant |i-j| \geqslant D$，这时也就没有必要精确计算 $d[i,j]$ 的值，而只需知道它们至少是 D。类似地，当 $|n_1-n_2| \geqslant D$ 的时候，我们也没有必要计算 S_1 和 S_2 的编辑距离。而当 $|n_1-n_2| < D$ 时，我们给出算法15。

算法 15 编辑距离的动态规划算法（带上界限制）

for $j \in \{0,1,\ldots,\min\{D,n_2\}\}$ **do**
  $d[0,j] \leftarrow j$
end for
for $i \in \{1,2,\ldots,n_1\}$ **do**
  $d[i,\max\{0,i-D\}] \leftarrow i-\max\{0,i-D\}$
  for $j \in \{\max\{0,i-D\}+1,\max\{0,i-D\}+2,\ldots,\min\{n_2,i+D\}\}$ **do**
    if $S_1(i)=S_2(j)$ **then**
      $d[i,j] \leftarrow d[i-1,j-1]$
    else
      $d[i,j] \leftarrow \min\{d[i,j-1],d[i-1,j],d[i-1,j-1]\}+1$
    end if
  end for

end for

注意到在算法15中，当 $d[i,j] \geqslant D$ 时，意味着编辑距离至少是 D，具体值并不一定准确。该算法将时间复杂度和空间复杂度均降为 $O(n_1 D)$。考虑到两个字符串的顺序可以交换，因此时间复杂度和空间复杂度均降为 $O(\min\{n_1, n_2\}D)$。这里我们同样可以使用滚动数组技巧，进一步将空间复杂度降至 $O(D)$。方法类似，读者可以自行尝试。

3.2.5 解的回溯

在前面几节中，我们主要关注如何计算 S_1 和 S_2 的编辑距离。在本节中我们进一步探讨如何获得相应的编辑过程。这实际上就是前面已经提到过的序列对齐。我们所要介绍的算法叫作 Needleman Wunsch 算法[NW70]。

假设我们已经通过算法13 得到了所有的 $d[i,j]$，则通过比较 $d[i,j]$ 和 $d[i-1,j-1]$、$d[i-1,j]$、$d[i,j-1]$，我们能够知道 $d[i,j]$ 是如何从“前一步”编辑得到的。

$$d[i,j] = \begin{cases} d[i-1,j-1], & \text{无编辑} \\ d[i,j-1]+1, & \text{插入} S_2(j) \\ d[i-1,j]+1, & \text{删除} S_1(i) \\ d[i-1,j-1]+1, & \text{将} S_1(i) \text{替换为} S_2(j) \end{cases} \tag{3.6}$$

根据算法13 的过程可知，$d[i,j]$ 的取值仅有这 4 种可能，也正好涵盖了 4 种可能的编辑方式。我们可以从 $d[n_1,n_2]$ 出发，反复使用式 (3.6)，判断它在上一步的基础上使用的是哪种编辑操作，以得到完整的编辑过程。

算法 16 编辑距离的回溯

1: $S_1^{\text{aligned}} \leftarrow$ “”。
2: $S_2^{\text{aligned}} \leftarrow$ “”。
3: $i \leftarrow n_1$。
4: $j \leftarrow n_2$。
5: **while** $i > 0$ 或 $j > 0$ **do**
6: **if** $j > 0$ 且 $d[i,j] = d[i,j-1]+1$ **then**
7: $S_1^{\text{aligned}} \leftarrow S_1^{\text{aligned}} +$ “-”。
8: $S_2^{\text{aligned}} \leftarrow S_2^{\text{aligned}} + S_2(j)$。
9: $j \leftarrow j-1$。

10: **else if** $i > 0$ 且 $d[i,j] = d[i-1,j] + 1$ **then**

11: $S_1^{\text{aligned}} \leftarrow S_1^{\text{aligned}} + S_1(i)$。

12: $S_2^{\text{aligned}} \leftarrow S_2^{\text{aligned}} +$ "-"。

13: $i \leftarrow i - 1$。

14: **else**

15: $S_1^{\text{aligned}} \leftarrow S_1^{\text{aligned}} + S_1(i)$。

16: $S_2^{\text{aligned}} \leftarrow S_2^{\text{aligned}} + S_2(j)$。

17: $i \leftarrow i - 1$。

18: $j \leftarrow j - 1$。

19: **end if**

20: **end while**

在算法16开始时，$i + j = n_1 + n_2$。在每一次循环后 i 和 j 中至少有一个减小 1，也就意味着 $i + j$ 至少减小 1。因此至多 $n_1 + n_2$ 次循环后，$i + j = 0$，即 i 和 j 都为 0，这也意味着算法16 的时间复杂度为 $O(n_1 + n_2)$。

3.2.6 分治算法

3.2.5节所介绍的 Needleman Wunsch 算法需要事先保存完整的 $d[i,j]$，这也就意味着 $O(n_1 n_2)$ 的空间复杂度。回忆我们前面提到过的滚动数组技巧，可以将动态规划降至线性空间复杂度，但代价是无法保存完整的 $d[i,j]$。那么是否存在一个算法仅有线性空间复杂度，却仍然能够求得完整的编辑过程？结论是肯定的。如果两个字符串中的某一个长度不超过 1，则直接用前面介绍过的方法即可。如果两个字符串的长度都超过 1，则我们可以设法拆分字符串，最终让其长度不超过 1。这种思想就是分治。

下面我们介绍 Hirschberg 算法[Hir75]，该算法使用了一个分治的框架。所谓分治就是要将大问题拆分成小问题分别处理。在编辑距离问题上，我们首先要考虑的就是如何将其拆分为小问题。一个很自然的拆分方法自然就是考虑子串的编辑距离。这也就涉及字符串的拆分。在此之前，我们先来定义几个与字符串拆分相关的符号。对于一个字符串 S，我们用 S^{L} 来表示 S 的某一个前缀，用 S^{R} 来表示 S 的某一个后缀，用 S^{-1} 来表示 S 的逆序串，用 $S^{-\text{R}}$ 来表示 S 的某一个后缀的逆序串，或者说是 S^{-1} 的某一个前缀。例如当 $S =$"abcde" 时，S^{L} 可以是 "a""ab""abc" 等，S^{R} 可以是 "e""de""cde" 等。$S^{-1} =$"edcba"，如果 $S^{\text{R}} =$"cde"，那么 $S^{-\text{R}} =$"edc"。两个字符串的编辑距离等于它们的逆序串的编辑距离，即 $d(S_1, S_2) = d(S_1^{-1}, S_2^{-1})$。

我们有这样一个定理。

定理 3.1 (分治的最优性) 任意将字符串 S_1 拆成前后两部分 $S_1^{\mathrm{L}}, S_1^{\mathrm{R}}$，对于字符串 S_2 都存在一种拆分 $S_2^{\mathrm{L}}, S_2^{\mathrm{R}}$，使得 $d(S_1, S_2) = d(S_1^{\mathrm{L}}, S_2^{\mathrm{L}}) + d(S_1^{\mathrm{R}}, S_2^{\mathrm{R}})$。

这个定理就为编辑距离的分治算法找到了一个依据。我们可以先将 S_1 拆分成 S_1^{L} 和 S_1^{R}，然后考虑如何拆分 S_2。一个很直观的观察是，将 S_1 编辑为 S_2 的过程实际上隐含了将 S_1^{L} 编辑为 S_2 的前半部分，以及将 S_1^{R} 编辑为 S_2 的后半部分。这也就是定理3.1成立的一个解释。如果我们使用动态规划去计算 $d(S_1^{\mathrm{L}}, S_2)$，则根据动态规划的算法流程可知，这实际上也就计算了 S_1^{L} 和 S_2 的每一个前缀的编辑距离。类似地，如果我们使用动态规划去计算 $d(S_1^{-\mathrm{R}}, S_2^{-1})$，这实际上也就计算了 S_1^{R} 和 S_2 的每一个后缀的编辑距离。如果我们把两个计算结果合并起来看，则通过遍历 S_2 的拆分方式，就可以找到 S_2^{L} 和 S_2^{R}，使得 $d(S_1^{\mathrm{L}}, S_2^{\mathrm{L}}) + d(S_1^{\mathrm{R}}, S_2^{\mathrm{R}})$ 达到最小。这时，我们也就知道了拆分 S_2 的最优方式。注意，这里 S_2^{L} 和 S_2^{R} 都可以是空串。

虽然拆分 S_1 的方式可以任意指定，但为了平衡分治算法的工作量，我们一般将 S_1 拆分成长度至多相差 1 的两个子串。下面我们给出算法核心，即函数 Hirschberg 的流程。注意到这个函数涉及递归，因此在每次递归调用时，输入参数不一定是原始字符串 S_1 和 S_2，所以我们用 $\bar{S}_1$、$\bar{S}_2$ 来代表输入参数。

算法 17 函数 Hirschberg($\bar{S}_1, \bar{S}_2$)

1: $\bar{S}_1^{\text{aligned}} \leftarrow$ ""。
2: $\bar{S}_2^{\text{aligned}} \leftarrow$ ""。
3: $\bar{n}_1 \leftarrow \text{length}(\bar{S}_1)$。
4: $\bar{n}_2 \leftarrow \text{length}(\bar{S}_2)$。
5: **if** $\bar{n}_1 = 0$ **then**
6: **for** $i \in \{1, 2, \ldots, \bar{n}_2\}$ **do**
7: $\bar{S}_1^{\text{aligned}} \leftarrow$ "-"。
8: **end for**
9: $\bar{S}_2^{\text{aligned}} \leftarrow \bar{S}_2$。
10: **else if** $\bar{n}_2 = 0$ **then**
11: **for** $i \in \{1, 2, \ldots, \bar{n}_1\}$ **do**
12: $\bar{S}_2^{\text{aligned}} \leftarrow$ "-"。
13: **end for**
14: $\bar{S}_1^{\text{aligned}} \leftarrow \bar{S}_1$。

15: **else if** $\bar{n}_1 = 1$ 或 $\bar{n}_2 = 1$ **then**

16: 调用算法13和算法16计算 $(\bar{S}_1^{\text{aligned}}, \bar{S}_2^{\text{aligned}})$。

17: **else**

18: $m_1 \leftarrow \lfloor \bar{n}_1/2 \rfloor$。

19: $\bar{S}_1^{\text{L}} \leftarrow \bar{S}_1(1 : m_1)$。

20: $\bar{S}_1^{\text{R}} \leftarrow \bar{S}_1(m_1 + 1 : \bar{n}_1)$。

21: 计算 $\bar{S}_1^{\text{L}}$ 和 $\bar{S}_2$ 的编辑距离。

22: 计算 $\bar{S}_1^{-\text{R}}$ 和 $\bar{S}_2^{-1}$ 的编辑距离。

23: 遍历 $\bar{S}_2$ 的拆分方式，找到 $\bar{S}_2^{\text{L}}$ 和 $\bar{S}_2^{\text{R}}$ 使得 $d(\bar{S}_1^{\text{L}}, \bar{S}_2^{\text{L}}) + d(\bar{S}_1^{\text{R}}, \bar{S}_2^{\text{R}})$ 达到最小。

24: $(\bar{S}_1^{\text{aligned}}, \bar{S}_2^{\text{aligned}}) \leftarrow \text{Hirschberg}(\bar{S}_1^{\text{L}}, \bar{S}_2^{\text{L}}) + \text{Hirschberg}(\bar{S}_1^{\text{R}}, \bar{S}_2^{\text{R}})$

25: **end if**

算法17 实际上生成了一个递归树。例如我们要计算 math 和 mouth 的编辑距离，则对应的递归树如图 3.2所示。首先我们将 math 拆分为 ma 和 th，可以算出 mouth 应当对应拆分为 mo 和 uth；然后进一步将 ma 拆分为 m 和 a，将 th 拆分为 t 和 h；在其中某一个字符串仅包含不超过一个字母时，递归终止。

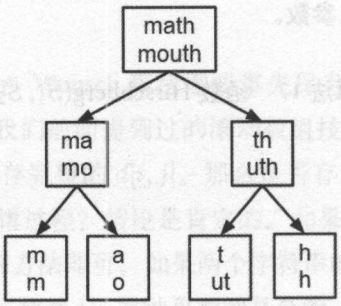


图 3.2 编辑距离的分治算法递归树示意图

在采用滚动数组优化后，算法17 的空间复杂度不会超过 $O(\min\{n_1, n_2\})$。下面我们来分析一下算法17的时间复杂度。时间复杂度主要集中在编辑距离的计算上。我们按照递归树的分层来考虑，记根节点为第 1 层，根节点的子节点为第 2 层，以此类推，在第 k 层的时候，S_1 已经被拆分过 $k-1$ 次，因此其长度应当是初始长度的 $1/2^{k-1}$ 倍，而同一层的所有 S_2 的长度总和应当是固定值保持不变，所以在第 k 层中计算编辑距离的时间复杂度为 $O(n_1 n_2/2^{k-1})$。将各层累加可知，算法17的时间复杂度是 $O(n_1 n_2)$，与单纯计算一次编辑距离的时间复杂度在同一量级。

3.2.7 多个字符串的编辑距离

除了两个字符串的编辑距离，在实际问题中可能还会涉及多个字符串的编辑距离。比如在拼写矫正中，当遇到一个字典中不存在的单词时，我们需要在字典中检查所有与其编辑距离接近的单词，并提示给用户。这时就需要知道输入的字符串与字典中所有字符串的编辑距离，或者说至少需要知道所有相近字符串的编辑距离。除了直接应用前面所介绍的方法，还有一些针对这类问题的特殊算法。这里我们简要介绍一下 BK-Tree 和 Levenshtein 自动机。

BK-Tree 是 [BK73] 提出的一个数据结构，以作者 Burkhard 和 Keller 的首字母命名。使用 BK-Tree 做拼写矫正，我们首先要将字典中的单词全部放入 BK-Tree 中。BK-Tree 初始为空，我们可以按照任意顺序插入字典中的单词。BK-Tree 中的每一个节点代表一个单词，每个节点可能有多个子节点。各个单词根据与当前节点的编辑距离，放入不同的子树中。这实际上类似于桶排序和哈希的思想。在查找的时候，查询词可以根据与当前节点的编辑距离来决定递归查找哪些子树，这样我们就快速排除了许多编辑距离过大的单词。

图 3.3 就是 BK-Tree 的一个示例。每一个方框代表字典中的一个字符串，每一条边标注了编辑距离。

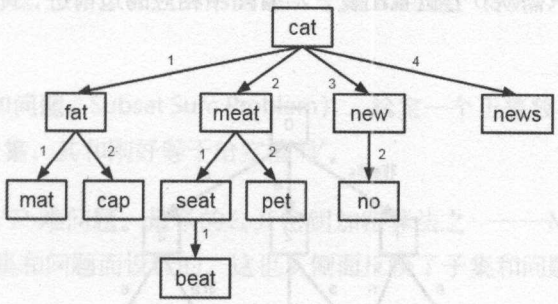


图 3.3 BK-Tree 示例

我们先来看如何找到字典中已有的字符串。以单词 beat 为例，beat 与 cat 的编辑距离为 2，于是我们找到 meat 所在的节点；beat 与 meat 的编辑距离为 1，于是我们找到 seat 所在的节点；beat 与 seat 的编辑距离为 1，于是沿着对应的边，我们找到了 beat。在实际使用中，我们需要找一个字符串的所有相似字符串。例如我们要找与字符串 now 的编辑距离不超过 1 的所有字符串。now 与 cat 的编辑距离为 3，不满足要求。根据三角不等式，meat、new、news 这三个分支都有可能包含与 now 的编辑距离不超过 1 的字符串。

- now 与 meat 的编辑距离为 4，不满足要求。根据三角不等式，这个分支不包含与 now 的编辑距离不超过 1 的字符串。
- now 与 new 的编辑距离为 1，因此我们找到了一个满足要求的字符串 new。根据三角不等式，no 这个分支也有可能包含与 now 的编辑距离不超过 1 的字符串。now 与 no 的编辑距离为 1，因此我们找到了第 2 个满足要求的字符串 no。
- now 与 news 的编辑距离为 2，不满足要求。

我们总共只计算了 5 次编辑距离就找到了全部满足要求的两个字符串，而在整个字典中包含 11 个字符串。这得益于 BK-Tree 帮我们快速排除了一些分支。

另外一种解决方案是 [SM02] 提出的 Levenshtein 自动机。与 BK-Tree 不同的是，Levenshtein 自动机并不是对字典建树，而是对查询词建树。对于任何一个查询词和一个给定的最大编辑距离，我们都可以建立一个 Levenshtein 自动机。对于字典中的词，如果能够在自动机中走到一个终止状态，就能知道它和查询词的编辑距离。如果无法走到终止状态，就说明它们的编辑距离过大，一般我们也就不关心了。

我们忽略建立自动机的过程，直接来看一下效果。图3.4 就是 Levenshtein 自动机的一个示例，它是对查询词 see 及最大编辑距离 1 而建立的。其中 0 号位置是初始位置，圆角方形的 10 号位置代表编辑距离为 0 的终止状态，圆形的 6 号和 9 号位置代表编辑距离为 1 的终止状态。任何一个词只需从 0 号位置出发，沿着图中相应的边前进，即可验证与 see 的编辑距离是否不超过 1。

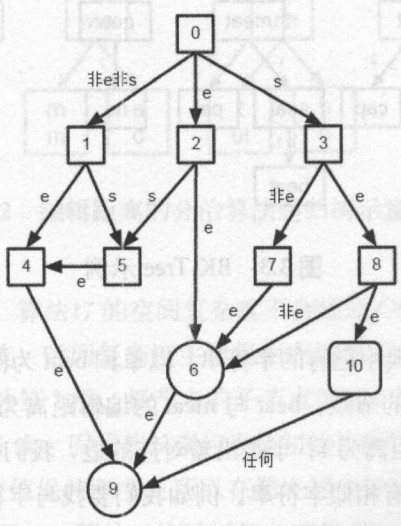


图 3.4 Levenshtein 自动机示例（查询词：see，最大编辑距离：1）

我们分别以 we、sea、see、seen 这 4 个词为例进行查询。

- we 对应的路径为：$0 \to 1 \to 4$，没有达到终止状态，因此编辑距离超过 1。
- sea 对应的路径为：$0 \to 3 \to 8 \to 6$，达到终止状态，因此编辑距离为 1。
- see 对应的路径为：$0 \to 3 \to 8 \to 10$，达到终止状态，因此编辑距离为 0。
- seen 对应的路径为：$0 \to 3 \to 8 \to 10 \to 9$，达到终止状态，因此编辑距离为 1。

可以看到，在建立好 Levenshtein 自动机后，检查一个单词的编辑距离可以在线性时间复杂度内完成。因此当需要计算许多词与同一个查询词的编辑距离时，这样做可以有效降低时间复杂度。如果事先将字典中的词都放入字典树（Trie），则在对查询词建立 Levenshtein 自动机后，只需计算字典树和 Levenshtein 自动机的交，就可以得到字典中所有与查询词接近的词，以及相应的编辑距离。

3.3 子集和问题

3.3.1 问题引入

在本节中我们考虑一个非常经典的问题：子集和问题。首先，我们来看这一问题的定义。

定义 3.2（子集和问题，Subset Sum Problem） 给定一个正整数集合 $\{w_1, w_2, \ldots, w_n\}$，判断是否存在一个子集，其和刚好等于给定值 W。

子集和问题是 $\mathcal{NP}$-难问题。最早的公开密钥加密算法之一——Merkle-Hellman 背包算法[MH78] 就是基于子集和问题而设计的。这也从侧面反映了子集和问题难于求解。

3.3.2 子集和问题的动态规划算法

我们定义 $s[i, j]$ 表示集合 $\{w_1, w_2, \ldots, w_i\}$ 是否存在一个子集，其和刚好等于 j。若存在则 $s[i, j]$ 为真，否则 $s[i, j]$ 为假。

算法 18 子集和问题的动态规划算法

1: $s[0, 0] \leftarrow$ 真。
2: **for** $i \in \{1, 2, \ldots, W\}$ **do**

3: $s[0,i] \leftarrow$ 假。
4: **end for**
5: **for** $i \in \{1,2,\ldots,n\}$ **do**
6: $s[i,0] \leftarrow$ 真。
7: **for** $j \in \{1,2,\ldots,w_i-1\}$ **do**
8: $s[i,j] \leftarrow s[i-1,j]$。
9: **end for**
10: **for** $j \in \{w_i, w_i+1,\ldots,W\}$ **do**
11: $s[i,j] \leftarrow s[i-1,j]$或$s[i,j-w_i]$。
12: **end for**
13: **end for**

算法18的时间复杂度和空间复杂度都是 $O(nW)$。注意到这里 W 是输入数值的大小。因此算法18是一个伪多项式算法，并不是多项式算法。

3.3.3 最优化问题

定义 3.3（子集和问题对应的最优化问题） 给定一个正整数集合 $\{w_1, w_2, \ldots, w_n\}$，求一个它的子集，其和不超过 W 且尽量大。可以归结为这样一个数学形式：

$$\begin{aligned} \max_{x} \quad & \sum_{i=1}^{n} w_i x_i, \\ \text{s.t.} \quad & \sum_{i=1}^{n} w_i x_i \leqslant W, \\ & x_i \in \{0,1\} \end{aligned} \tag{3.7}$$

显然，求解子集和问题对应的最优化问题比求解子集和问题本身更难。因为它不仅能回答“是”与“否”，在“否”的时候还能给出一个近似解。同样，子集和问题对应的最优化问题也存在动态规划解法。

算法 19 子集和问题对应的最优化问题的动态规划算法

1: **for** $i \in \{0,1,2,\ldots,W\}$ **do**
2: $s[0,i] \leftarrow 0$。
3: **end for**
4: **for** $i \in \{1,2,\ldots,n\}$ **do**

```
    s[i, 0] ← 0。
    for j ∈ {1, 2, ..., w_i − 1} do
        s[i, j] ← s[i − 1, j]。
    end for
    for j ∈ {w_i, w_i + 1, ..., W} do
        s[i, j] ← max{s[i − 1, j], s[i, j − w_i] + w_i}。
    end for
end for
```

由于子集和问题对应的优化问题含义更广泛，也更为实用，因此我们经常不加区分地将这一问题直接称为子集和问题。

3.3.4 滚动数组的技巧

类似编辑距离的问题，在这里使用滚动数组的技巧，我们可以将空间复杂度降到 $O(W)$。

算法 20 子集和问题的动态规划算法（使用滚动数组技巧优化）

```
for i ∈ {0, 1, 2, ..., W} do
    s^old[i] ← 0。
end for
for i ∈ {1, 2, ..., n} do
    s^new[0] ← 0。
    for j ∈ {1, 2, ..., w_i − 1} do
        s^new[j] ← s^old[j]。
    end for
    for j ∈ {w_i, w_i + 1, ..., W} do
        s^new[j] ← max{s^old[j], s^new[j − w_i] + w_i}。
    end for
    交换 s^old 和 s^new。
end for
```

3.3.5 贪婪算法

一类简单而有效的近似算法就是贪婪算法（Greedy Algorithm）。对于子集和问题而言，一个很直接的想法就是尽量先挑大的数放入子集，这样能尽快靠近目标值 W，直到无法放入新的物品为止。由于逻辑十分简单，所以这样的算法在计算速度上一定是非常有优势的。

算法 21 子集和问题的贪婪算法（大数优先）

1: 排序，使得 $w_1 \geqslant w_2 \geqslant \cdots \geqslant w_n$。
2: $s \leftarrow 0$。
3: **for** $i \in \{1, 2, \ldots, n\}$ **do**
4: **if** $s + w_i \leqslant W$ **then**
5: $s \leftarrow s + w_i$。
6: **end if**
7: **end for**

由于需要排序，算法的时间复杂度为 $O(n \log n)$，空间复杂度为 $O(1)$。

定理 3.2 算法21的近似比为 0.5。

证 当第 1 次遇到 $s+w_i > W$ 时，由于排序的原因，我们有 $s \geqslant w_i$，因此 $2s \geqslant s+w_i > W$，故 $s > W/2$。s 在程序执行的过程中只可能增加，不可能减少，所以在程序结束时必然也有 $s > W/2$。得证。 □

值得一提的是，在最坏的情况下，贪婪算法的计算结果确实可能会非常接近 0.5 这一近似比。例如对于仅包含 3 个元素的集合 $\{W/2+1, W/2, W/2\}$，贪婪算法得到的子集和是 $W/2+1$，而最优值显然是 W，近似比为 $(W+2)/(2W)$。由于 $\lim\limits_{W\to\infty} (W+2)/(2W) = 1/2$。所以当 W 很大时，这一近似比将会非常接近 0.5。

反过来，我们优先选择较小的数会如何呢？

算法 22 子集和问题的贪婪算法（小数优先）

1: 排序，使得 $w_1 \leqslant w_2 \leqslant \cdots \leqslant w_n$。
2: $s \leftarrow 0$。
3: **for** $i \in \{1, 2, \ldots, n\}$ **do**
4: **if** $s + w_i \leqslant W$ **then**

5: $s \leftarrow s + w_i$。
6: **end if**
7: **end for**

这样一个非常类似的算法却会差很多。例如对于 $\{1, W\}$ 这样一个简单的集合，算法得到的子集和是 1，而最优值是 W，近似比为 $1/W$。当 W 很大时，这一近似比会接近 0。也就是说这一算法的效果完全无法保证。

3.3.6 松弛动态规划

一个完美的算法应该具备的优势有：计算速度快、占用内存小、计算结果精确。对于大规模问题，我们很难让一个算法同时满足以上三点。前面提到近似算法的优势是计算速度快、占用内存小；动态规划算法的优势是计算结果精确。因此如果能取这二者之长，就有可能得到一个更好的算法。

下面我们考虑一种近似的动态规划算法——松弛动态规划，它是“近似的”动态规划方法。对于大规模问题，它可以有效地避免状态空间爆炸。其思想是将状态空间进行划分，对于相似的状态，我们仅记录其中的一部分。这样做的好处是：我们依然维护了一些状态，因此计算结果近似正确；对于相似的状态，我们只记录了其中一些“代表”，有效降低了内存的占用和计算量。

本节介绍的算法来自于 [KMPS03]。

我们将左开右闭区间 $(0, W]$ 等分为 $m = \dfrac{1}{\epsilon}$ 个左开右闭的小区间：$I_1 = (0, \epsilon W]$，$I_2 = (\epsilon W, 2\epsilon W], \cdots, I_m = (W - \epsilon W, W]$，每个区间的长度都是 ϵW。如果有多个值落在同一个小区间中，则我们仅记录其中两个值：最小值和最大值。原本子集和有 W 个可能性 $1, 2, \cdots, W$，现在我们考虑了 $\dfrac{1}{\epsilon}$ 个小区间之后，只需要记录其中 $\dfrac{2}{\epsilon}$ 个可能性。这里要额外说明一点，对于一个数 $v \in (0, W]$，我们只需要计算 $\dfrac{v}{\epsilon W}$ 就能知道它属于哪一个小区间，这一点在算法过程中非常有用。

下面我们就用 $s^-[j]$ 和 $s^+[j]$ 来表示第 j 个小区间的最小值和最大值。

算法 23 子集和问题的松弛动态规划算法（计算近似最优值）

1: **for** $i \in \{1, 2, \ldots, n\}$ **do**
2: $\Delta \leftarrow \{s^-[j] + w_i, s^+[j] + w_i, w_i\}_{j=1}^{m} \cap (0, W]$。
3: **for** $\delta \in \Delta$ **do**

```
4:      找到 δ 所在的小区间为 I_j。
5:      if δ < s⁻[j] then
6:          s⁻[j] ← δ。
7:      end if
8:      if δ > s⁺[j] then
9:          s⁺[j] ← δ。
10:     end if
11:   end for
12: end for
13: 输出 W̃ ← max{s⁺[1], s⁺[2], ..., s⁺[m]}。
```

直观的感觉是，每一次循环都丢掉了许多信息，很可能会得到一个比较差的结果。但令人意外的是，这个算法可以达到 $1-\epsilon$ 的近似比，甚至在某些情况下还能精确求解。具体来说，我们有如下定理成立。

定理 3.3 算法23的输出结果 $\tilde{W}$ 与子集和问题的最优解 W^* 至少满足下面一种情况。

- $\tilde{W}=W^*$。
- $\tilde{W}\in[W-\epsilon W, W]$。

在定理3.3给出的两种情况中，第 1 种情况是说我们求得了最优值，第 2 种情况是说我们不一定求得最优值，但距离最优值的上界 W 至多相差 ϵW，算法的近似比是 $1-\epsilon$，非常接近 1。这两种情况都说明我们找到了非常好的最优值。

算法23的时间复杂度为 $O(n/\epsilon)$，空间复杂度为 $O(n+1/\epsilon)$。我们可以自由地调整 ϵ 的大小，在时间空间和近似比之间找到合适的平衡点。

同动态规划算法一样，松弛动态规划算法也可以利用分治策略来回溯最优解，只是稍显复杂。

3.3.7 相关问题

子集和问题也有一个特殊情况：当 $W=\dfrac{\sum_{i=1}^{n} w_i}{2}$ 的时候，它变为划分问题。

定义 3.4（划分问题，Partition Problem） 将集合 $\{w_1, w_2, \ldots, w_n\}$ 划分为两个部分，使得两部分的和相等。

划分问题只是子集和问题的一个特例，因此子集和问题的算法可以直接用来解划分问题。反过来划分问题的算法也可以直接解子集和问题。对于任意给定的 W，我们构造集合 $\{w_1, w_2, \ldots, w_n, 2W - \sum_{i=1}^{n} w_i\}$，并求解划分问题。在最终结果中应当有一个子集不包含 $2W - \sum_{i=1}^{n} w_i$，且子集和为 W。

与划分问题类似的还有 3-划分问题（3-Partition Problem）。

定义 3.5（3-划分问题，3-Partition Problem） 在给定集合中包含 $3m$ 个元素，将集合划分为 m 个部分，每个部分刚好包含 3 个元素，且和相等。

3-划分问题经常被错误地认为是将集合划分为 3 个部分，使得它们的和相等。而事实上 3-划分问题要比这难得多，是强 $\mathcal{NP}$-难问题。

子集和问题可以视作背包问题的特例。背包问题实际上包含非常多的变种，在这里我们仅考虑 0-1 背包问题。

定义 3.6（0-1 背包问题） 假设一个人有 n 个物品可以放入背包，其中第 i 个物品的重量为 w_i，价值为 v_i。背包的最大承重是 W。目标是在背包的承重范围内，装入总价值尽可能大的物品。

0-1 背包问题可以写成这样一个最优化问题：

$$\begin{aligned} \max_{x} \quad & \sum_{i=1}^{n} v_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^{n} w_i x_i \leqslant W \\ & x_i \in \{0, 1\} \end{aligned} \tag{3.8}$$

0-1 背包问题有一个特殊情况：在不考虑物品价值，或者说物品的价值和重量相同时，它变为子集和问题。

0-1 背包问题有经典的动态规划算法。我们定义 $s[i, k]$ 表示仅从前 i 个物品中选择一些放入背包，在装入总重不超过 k 的物品时所能达到的最大总价值。我们最终要的答案就是 $s[n, W]$。我们有如下初始状态：

$$s[0, k] = 0 \tag{3.9}$$

这是因为我们不考虑任何一个物品，因此背包为空，所能达到的最大总价值是 0。

$$s[i,k]=\begin{cases}\max\{s[i-1,k],v_i+s[i-1,k-w_i]\}, & k\geqslant w_i\\ s[i-1,k], & k<w_i\end{cases} \tag{3.10}$$

在这个递推式中，我们考虑处理第 i 个物品的可能性。如果我们不将第 i 个物品放入背包，则状态转移为 $s[i-1,k]$，否则状态转移为 $s[i-1,k-w_i]$。这一动态规划算法的时间复杂度为 $O(nW)$，空间复杂度为 $O(nW)$。与子集和问题类似，0-1 背包问题也有许多优化技巧及近似算法，这里就不一一介绍了。

3.4 旅行商问题

3.4.1 问题引入

旅行商问题是图论中的一个经典问题。其问题背景是：有一个商人，他要从某一个城市出发，走访若干个指定城市刚好各一次，最终回到原来所在的城市。我们需要为其设计一条路线，使得总费用最少。这里的费用可以认为是金钱，也可以认为是时间或者距离等。由于最终要回到出发点，因此具体从哪里出发并不重要，不影响最优路线的选择。我们将旅行商问题描述如下。

定义 3.7（旅行商问题，Travelling Salesman Problem，TSP） 给定一些城市及每两个城市之间的距离，求一条回路经过每个城市刚好各一次，并且其长度最短。

为了简单起见，我们假设任意两个城市之间都是可达的。即任意选定两个城市，都可以找到一个从一个城市到达另一个城市的方式。这是一个合理的假设，如果某两个城市之间不可达，也就没有必要把它们放在一起讨论。

旅行商问题的判定形式是，给定一个距离 L，是否存在一个回路，使其长度小于 L。这一判定问题是 $\mathcal{NP}$-难问题。旅行商问题比前面的子集和问题要更难，子集和问题是 $\mathcal{NP}$-难问题，而旅行商问题是强 $\mathcal{NP}$-难问题。

我们记城市的数量为 n，n 个城市分别记为 $c_1,c_2,\cdots,c_n$，城市 c_i 到 c_j 的距离为 $d(c_i,c_j)$。不失一般性，我们假设出发点为 c_1。

常见的旅行商问题还包括以下几种类型。

- 满足三角不等式的旅行商问题（Metric TSP）：如果对于任意三个城市 c_i、c_j、c_k，则总有 $d(c_i, c_j) \leqslant d(c_i, c_k) + d(c_k, c_j)$。
- 欧几里得旅行商问题（Euclidean TSP）：任意两个城市间的距离就是欧几里得距离。欧几里得旅行商问题也是满足三角不等式的旅行商问题。
- 对称旅行商问题（Symmetric TSP）：对于任意两个城市 c_i, c_j，我们有 $d(c_i, c_j) = d(c_j, c_i)$。相应地，也有非对称旅行商问题（Asymmetric TSP）。

这几类问题在原本的旅行商问题的基础上增加了一些限制，使得问题看起来更简单。虽然这些问题也是难于求解的，但相对来说更容易设计算法。

旅行商问题有许多推广，使其更接近实际问题。其中一个推广就是多旅行商问题（Multiple TSP）。在多旅行商问题中，多个旅行商从同一个指定城市出发，各个旅行商都经过一些城市并回到出发点，使得每个城市都刚好被经过了一次，要求给出路线使得总费用最小。这在物流管理中是很重要的一个问题。因为安排车辆派送货物给若干指定客户，所以与多旅行商问题非常类似。如果我们再进一步考虑每一辆车的运输能力，则可以进一步得到车辆路径问题（Vehicle Routing Problem）。

求解旅行商问题最直接的方式就是穷举法，即枚举 n 个城市的所有排列方式，计算依次访问的总距离，找到最短距离。但排列方式总共有 $n!$ 个，这就意味着即使只有 20 个城市，排列方式就有 $20! \approx 2.4 \times 10^{18}$。因此穷举法的运行时间是无法接受的。

这里还要顺便提及一些与旅行商问题相关的几个重要概念，这几个概念在后面会被用到。

定义 3.8（哈密顿路，Hamiltonian Path） 经过每个顶点刚好一次的路。

定义 3.9（哈密顿回路，Hamiltonian Cycle） 经过每个顶点刚好一次的回路。

旅行商问题实际上就是求最短的哈密顿回路。类似地，求最短的哈密顿路同样是非常难的问题。

3.4.2 动态规划算法

本节我们介绍Held-Karp算法，又称Bellman Held Karp算法，是由两篇文章[Bel62, HK62]独立提出的。

首先我们给出一些记号。记集合 $S \subseteq \{c_2, c_3, \cdots, c_n\}$，$d[S, c_i]$ 表示从 c_1 出发经过 S 中的所有城市且只经过这些城市，最后抵达 c_i 的最短路径长度，其中 $c_i \in S$，则我们得到以下几个性质。

- $d[\{c_i\}, c_i] = d(c_1, c_i)$。这个简单的性质可以用于初始化集合 S 仅包含1个元素的情况。
- $\min\limits_{2 \leqslant i \leqslant n} \{d[\{c_2, c_3, \cdots, c_n\}, c_i] + d(c_i, c_1)\}$ 是最终要求的旅行商问题的最短路径长度。
- $d[S, c_i] = \min\limits_{c_j \in S - \{c_i\}} \{d[S - \{c_i\}, c_j] + d(c_j, c_i)\}$。这个式子给出了状态转移规则。这个状态转移过程就是在枚举 c_i 的前一个城市 c_j。

根据以上分析，我们可以得到旅行商问题的动态规划算法。

算法 24 旅行商问题的动态规划算法

```
for i ∈ {2, ··· , n − 1} do
    d[{c_i}, c_i] ← d(c_1, c_i)。
end for
for i ∈ {2, ··· , n − 1} do
    for S ⊆ {c_2, ··· , c_n}且|S| = i do
        for c_j ∈ S do
            d[S, c_j] ← min_{c_k ∈ S − {c_j}} {d[S − {c_j}, c_k] + d(c_k, c_j)}。
        end for
    end for
end for
```

算法24的时间复杂度是 $O(n^2 2^n)$，空间复杂度是 $O(n2^n)$。虽然这一算法仍旧只能解决非常小规模的问题，但与穷举法相比有所改进。

3.4.3 一笔画问题

一笔画问题是与旅行商问题非常相关的一个问题。本节我们介绍与一笔画问题相关的概念和算法，用来启发我们得到旅行商问题的一些新思路。

在旅行商问题中，我们要求经过每个顶点刚好一次。如果不是经过每个顶点，而是改为经过每条边，则我们也有相应的概念。

定义 3.10（欧拉路，Eulerian Path） 经过每条边刚好一次的路。

定义 3.11（欧拉回路，Eulerian Cycle） 经过每条边刚好一次的回路。

注意欧拉路和欧拉回路都是经过每条边刚好一次，所以欧拉路的长度和欧拉回路的长度都是所有边长的和，也就不存在最短欧拉路和最短欧拉回路的问题。我们一般关心的问题是，对于任意一个给定的图是否存在欧拉路和欧拉回路，这也就是所谓的一笔画问题。与旅行商问题类似，对于一笔画问题我们同样假设在任意两个城市之间是可达的。

一笔画问题有这样几个很有趣的结论。

- 一个无向图有欧拉回路当且仅当每个顶点的度数都是偶数。
- 一个无向图有欧拉路当且仅当有至多 2 个顶点的度数是奇数。
- 一个有向图有欧拉回路当且仅当每个顶点的入度和出度相等。
- 一个有向图有欧拉路当且仅当有至多一个顶点的入度比出度大 1，至多一个顶点的出度比入度大 1，且其余顶点的入度和出度相等。

Hierholzer 算法[HW73] 可以在线性时间内求得一个欧拉路或欧拉回路。算法思路非常简单，我们以欧拉回路为例给出说明。假设我们的图满足前面所描述的条件，存在欧拉回路。我们以任意顶点 u 作为出发点，沿着尚未经过的路一直走，直到走回 u，得到一个回路。这一过程不会卡在其他顶点，因为顶点的度数决定了一旦走到这一顶点，则必然有尚未走过的边可以出来。但这一过程可能有顶点和边没有走过。此时一定能在当前的回路上找到一个顶点 v，使得 v 有边没有走过。这时再从 v 出发，类似前面的过程再得到一个新的回路，并与之前的回路合并为同一条回路。最终，我们可以得到一个欧拉回路。

对于满足三角不等式的旅行商问题，欧拉回路可以用来构造哈密顿回路，这个思路有助于我们设计旅行商问题的近似算法。对于已知的一个欧拉回路，我们可以将其按照经过顶点的先后顺序，去除重复出现的顶点（终点除外），得到相应的哈密顿回路。例如欧拉回路经过的顶点顺序为 $1, 2, 3, 2, 4, 5, 3, 1$，我们去除重复出现的顶点后，得到 $1, 2, 3, 4, 5, 1$。这样构造得到的哈密顿回路由于跳过了一些顶点，根据三角不等式，其总长度会小于欧拉回路。这一结论告诉我们，如果能够构造出总长度很短的欧拉回路，我们就能够得到总长度更短的哈密顿回路。而欧拉回路的构造算法是非常简单的，因此我们可以得到一个简单的方法来构造哈密顿回路。

3.4.4 Christofides 算法

本节我们考虑满足三角不等式的对称旅行商问题。在对称旅行商问题中所考虑的图是无向图。注意到旅行商问题要求的是最短回路，如果我们放松要求，则不要求形成回路而仅仅要求连接图中的所有点，那就和生成树非常类似了。我们先来看生成树和最小生成树的定义。

定义 3.12（生成树，Spanning Tree） 对于一个包含 n 个顶点的连通无向图，如果仅选取其中 $n-1$ 条边，使得任意两点可达，则称所选边是该图的生成树。

定义 3.13（最小生成树，Minimum Spanning Tree） 对于一个连通无向图，其总边长最小的生成树叫作最小生成树。

对于对称旅行商问题来说，我们只考虑连通无向图。对于给定的图 G，我们记其最小生成树的长度为 $d_G^*(\mathrm{MST})$，其旅行商问题的最优值为 $d_G^*(\mathrm{TSP})$。对于旅行商问题的最优解来说，任意删除一条边都会得到一个生成树，因此最小生成树的长度应当严格小于旅行商问题的最优值。这两者之间的关系应当满足

$$d_G^*(\mathrm{MST}) < d_G^*(\mathrm{TSP}) \tag{3.11}$$

反之，对于最小生成树来说，我们可以复制它的每一条边，再根据前面一笔画问题的性质，我们就可以构造一条长度为 $2d_G^*(\mathrm{MST})$ 的欧拉回路，从而进一步构造哈密顿回路。这条哈密顿回路不一定是最短的，因此长度不小于 $d_G^*(\mathrm{TSP})$。所以我们有

$$d_G^*(\mathrm{TSP}) \leqslant 2d_G^*(\mathrm{MST}) \tag{3.12}$$

图 3.5 给出了一个双最小生成树的示例，图的左边是一个最小生成树。我们在复制了它的每条边后就可以构造出欧拉回路。我们可以利用这个特点构造一个近似算法：双最小生成树算法（Double Minimum Spanning Tree Algorithm）。

算法 25 双最小生成树算法

1: 构造最小生成树。
2: 将最小生成树的每条无向边变成两个方向相反的有向边，并对这些有向边构造欧拉回路。
3: 根据欧拉回路构造长度更短的哈密顿回路。

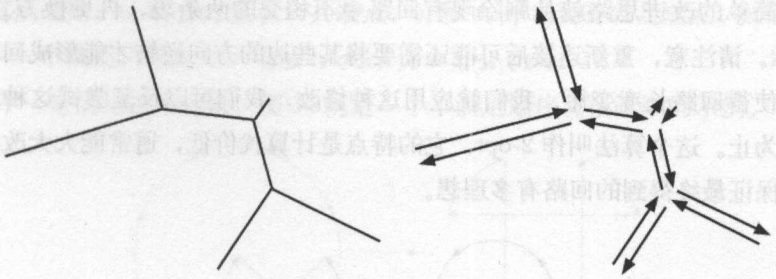


图 3.5　双最小生成树示例

双最小生成树算法得到的哈密顿回路长度应当不超过哈密顿回路最优值的两倍，因此近似比是 2。

我们可以看到，在双最小生成树算法中，最影响近似比的一步是将每条无向边变成两个方向相反的有向边。如果要改进这一近似比，则我们可以尝试用其他方式得到更短的欧拉回路。根据欧拉回路存在的条件，我们其实只要将度数为奇数的边都变成偶数即可。这一目标可以看作最小完美匹配问题，使用带花树算法（Blossom Algorithm）在多项式时间内求解。这样我们就可以得到 Christofides 算法。

算法 26　Christofides 算法

1: 构造 G 的最小生成树 T。
2: 记 O 是最小生成树 T 中所有度为奇数的顶点的集合，找到 O 的一个最小完美匹配 M。
3: 构造一个欧拉回路只经过 T 和 M 中的边。
4: 根据欧拉回路构造长度更短的哈密顿回路。

因为由哈密顿回路中一半的边就可以构成一个匹配，但不一定是最小匹配，所以最小完美匹配的长度应当不超过旅行商问题最优值的一半。因此 Christofides 算法的近似比是 3/2。

3.4.5　Lin-Kernighan 算法

前面介绍的都属于构造算法，即根据某种方式直接构造回路。本节将要介绍一个迭代算法，我们可以从任意一个初始回路出发，通过修改部分路径使得回路总长度变小。迭代算法可以用于改进任何一个已有的回路，自然包括所有构造算法得到的回路。因此一个常见的方案是先利用某种构造算法求得一个近似回路，再利用迭代算法对其进行改进。

一个最简单的改进思路就是删除现有回路中不相交的两条边，再更换方式重新连接，如图 3.6所示。请注意，重新连接后可能还需要将某些边的方向逆转才能形成回路。如果这种修改可以使得回路长度变短，我们就应用这种修改，我们可以反复尝试这种策略直到无法继续改进为止。这个算法叫作 2-opt，它的特点是计算代价低，通常能大大改进现有的回路，但很难保证最终得到的回路有多理想。

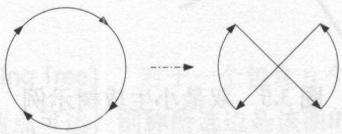


图 3.6 2-opt 示意图

类似 2-opt，我们还有 3-opt。3-opt 思想与 2-opt 非常类似，区别是即使选定了删除的边，我们也可以有多种方式添加边。如图 3.7所示，我们给出了两种添加边的方式。因此在 3-opt 中，我们需要尝试的选择要比 2-opt 多很多。

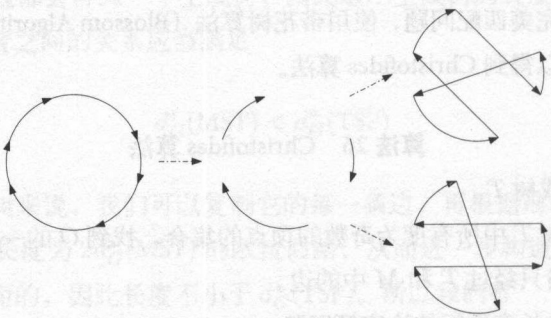


图 3.7 3-opt 示意图

类似地，我们还可以有 4-opt、5-opt 等，但是这样继续推广会导致添加边的方式呈指数级增加（请思考原因）。这样得到的算法会因为计算量过大而不再实用。如果我们希望使用 4-opt、5-opt，甚至任意的 r-opt，却不希望让添加边的方式呈指数级爆炸，则一种简单的思路就是设计某种规则来限制添加边的方式。

Lin-Kernighan 算法[LK73] 就采用了一种新的方式进行推广，在计算量合理的情况下，有更大的可能性得到最优解。我们记集合 $X = \{x_1, x_2, \ldots, x_r\}, Y = \{y_1, y_2, \ldots, y_r\}$，假设从当前回路中删掉集合 X 中的边，替换为集合 Y 中的边，则可以得到更短的回路。这实际上就是 r-opt 算法。在 Lin-Kernighan 算法中，我们定义若干准则，限制 X 和 Y 的选取，合理控制运算量。

- 顺序交换准则。x_i 和 y_{i+1} 要有公共端点，y_i 和 x_{i+1} 也要有公共端点。如果记 p_1 是 x_1 的其中一个端点，则有 $x_i = (p_{2i-1}, p_{2i})$, $y_i = (p_{2i}, p_{2i+1})$。

  r-opt 不一定满足这个性质。图 3.8就是一个不满足顺序交换准则的示例。

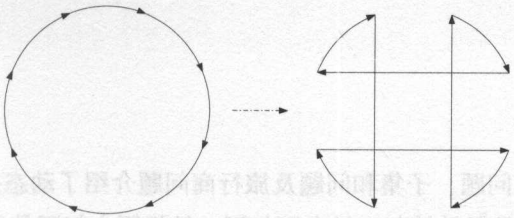


图 3.8 不满足顺序交换准则的示例

- 可行性准则。如果我们将 y_i 选为 (p_{2i}, p_1)，则可以得到一个回路。这样的性质可以保证我们每选择一个新的 x_i 之后，就可以尝试将 y_i 选为 (p_{2i}, p_1)，如果在这种选择下用 Y 替换 X 能够改进回路长度，就可以类似 r-opt 那样改进回路。
- 改进准则。每次选择 y_i 时，都保证 $y_1, y_2, \cdots, y_i$ 的总长度要小于 $x_1, x_2, \cdots, x_i$ 的总长度。

算法27给出了 Lin-Kernighan 算法的核心部分。每调用一次算法27，都会尝试对给定图及给定回路进行一次改进。如果改进回路成功，则可以重复调用并继续改进。

算法 27 Lin-Kernighan 算法核心部分

输入: 图 $G(V, E)$，初始回路 T。

```
for p_1 ∈ V do
  for x_1 = (p_1, p_2) ∈ T do
    for y_1 = (p_2, p_3) ∉ T 满足改进准则 do
      i ← 2。
      repeat
        选择 x_i = (p_{2i-1}, p_{2i}) ∈ T 满足可行性准则。
        如果选择 y_i = (p_{2i}, p_1)，再将 X 替换为 Y 可以改进回路，则程序结束。
        选择 y_i = (p_{2i}, p_{2i+1}) ∉ T 满足改进准则。
        i ← i + 1。
      until 无法选择 x_i, y_i
    end for
  end for
end for
```

在算法27中，我们实际上遍历了 x_1, y_1 的所有选择，但没有遍历其他边的所有选择。如果实际计算时间允许，则也可以适当修改，遍历更多选择。更多的遍历意味着更长的计算时间和更好的计算结果。关于 Lin-Kernighan 的高效实现，可以进一步参考 [Hel00]。

3.5 总结

我们通过编辑距离问题、子集和问题及旅行商问题介绍了动态规划与近似算法的基本思想和应用。这三个问题的计算复杂性各不相同，编辑距离问题是 $\mathcal{P}$-问题，子集和问题是 $\mathcal{NP}$-难问题，旅行商问题是强 $\mathcal{NP}$-难问题。这三个问题虽然都能够利用动态规划求解，但对于越难的问题，动态规划能解决的问题规模越小。相反，近似算法对于越难的问题则意义越大。尤其是旅行商问题，动态规划通常仅能计算 20 个城市左右，而近似算法使得我们可以处理数万个城市的问题。我们在处理其他复杂的大规模问题时，也应当充分考虑动态规划和近似算法的结合，使我们所设计的算法在满足计算精度要求的前提下也能满足计算效率和内存占用的要求。

第4章

高斯消去法

4.1 问题引入

线性方程组是最简单也是最重要的一类代数方程组。大量的科学技术问题最终往往归结为解线性方程组，因此线性方程组的数值解法在计算数学中占有重要的地位。在本章中我们将介绍求解线性方程组的方法之一——高斯消去法。我们将线性方程组写成矩阵的形式，例如 $\boldsymbol{A}\boldsymbol{x}=\boldsymbol{b}$，其中 $\boldsymbol{A}$ 叫作系数矩阵，$\boldsymbol{b}$ 叫作右端项，$\boldsymbol{x}$ 叫作未知向量。

我们在求解方程组前，必须先了解方程组是否有解及是否存在唯一解。线性方程组有时会存在多解或者无解的情况，这主要来源于冗余方程和冗余变量。例如方程 $2x_1+3x_2=1$ 和方程 $4x_1+6x_2=2$ 实际上是一回事，其中一个方程就可以被认为是冗余的。稍微复杂一点，如果有两个方程分别是 $x_1+x_2+x_3=1$ 及 $x_2+2x_3=2$，第 3 个方程是 $x_1+2x_2+3x_3=3$，那么第 3 个方程就是冗余的，因为只要前两个方程成立，将它们相加就可以得到第 3 个方程。如果将第 3 个方程改为 $x_1+2x_2+3x_3=4$，就会与前两个方程矛盾，必然导致无解。这种方程冗余反映在数学概念上就是矩阵 $\boldsymbol{A}$ 不是行满秩的。冗余变量也是类似的情况，例如仅包含一个方程的方程组 $x_1+x_2=1$，这两个变量的取值都是不定的，它们之中有一个变量是冗余的，如果能固定住其中一个变量，则另一个变量也就可以随之确定下来。这种变量冗余反映在数学概念上就是矩阵 $\boldsymbol{A}$ 不是列满秩的。如果方程冗余，则方程组可能无解。如果变量冗余，则方程组可能多解。

矩阵 $\boldsymbol{A}$ 的行数和列数的关系也能反映解的情况，尽管并不精确。

- 如果 $\boldsymbol{A}$ 的行数小于列数，也就是说方程的个数小于变量的个数，那么这时我们称方

程组是欠定的。对于欠定方程组，或者无解，或者有无穷多个解。

- 如果 $\boldsymbol{A}$ 的行数大于列数，也就是说方程的个数大于变量的个数，那么这时我们称方程组是超定的。超定方程组很可能是无解的。
- 如果 $\boldsymbol{A}$ 的行数等于列数，也就是说方程的个数等于变量的个数，那么这时方程组常常有唯一解。

在本章中，我们为了表述方便，仅考虑 A 的行数和列数相等且有唯一解的情况，但介绍的算法仍然可以推广到其他情况。

从纯数学的角度来看，线性方程组的解就是 $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$。看上去非常简单，只要计算 $\boldsymbol{A}^{-1}$ 就可以了。确实是这样，但又不仅仅是这样。在实际问题中遇到的线性方程组往往 $\boldsymbol{A}$ 的规模超大，连存储都是问题，更不要说求解。如何灵活地利用问题特性及自己能够支配的计算资源更快、更准地求解，才是问题的关键。在面对线性方程组时，一个常见的误区就是去研究矩阵求逆，认为提高了矩阵求逆的速度就可以提高线性方程组的求解速度。可惜这通常不是一个正确的方向。目前流行的求解线性方程组的方法包括两大类：迭代法和直接法。迭代法是指从某个初始向量出发，逐步对其进行调整，最终得到方程的解；直接法则不使用迭代的方式，通过对问题的一些操作，直接得到解。这两类方法各有千秋。最常见的直接法就是本章将要介绍的高斯消去法。一般来说，迭代法对于大规模的线性方程组更为有效，也比直接法更容易处理精度问题。既然如此，为什么我们还要关注直接法？有一类需求是要求解一系列右端项不同，但系数矩阵相同的线性方程组。这时对于直接法而言，在求解第 1 个线性方程组后，其余的都可以借助之前的计算结果很快地得到解。而对于迭代法而言，每一个线性方程组都要重新来过。这时直接法相比迭代法更有优势。

4.2 矩阵编程基础

一维数组对应数学中的向量，二维数组对应数学中的矩阵，这是非常自然的想法。使用二维数组存储矩阵会有一些比较细节的问题。比如不同的编程语言对二维数组的实际存储方式会有所不同。在 C/C++ 中，二维数组采用行优先（Row-Major）存储，而 Fortran 语言则是列优先（Column-Major）存储。也就是说，矩阵

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{pmatrix} \tag{4.1}$$

在C语言中的存储顺序是 $a_{1,1}, a_{1,2}, a_{1,3}, a_{2,1}, a_{2,2}, a_{2,3}$，而在Fortran语言中的存储顺序是 $a_{1,1}, a_{2,1}, a_{1,2}, a_{2,2}, a_{1,3}, a_{2,3}$。无论是行优先存储还是列优先存储，在某些情况下可能会对计算效率产生影响。

在实际编程中，我们往往选用一维数组存储矩阵，自己定义存储顺序。这样做有以下几个好处。

- 可以自由决定行优先或列优先。在不同的情况下，指定行优先或列优先可以为算法加速。
- 仅需一次动态分配内存。二维数组动态分配较为麻烦，也会稍微影响效率。
- 一维数组更便于作为参数传递。

对于一个 m 行 n 列的矩阵，在行优先时，$a_{i,j}$ 的存储位置是 $i \times n + j$；在列优先时，$a_{i,j}$ 的存储位置是 $j \times m + i$。

在编写矩阵计算的程序时，应当尽量避免跳跃访问矩阵中的元素，而尽可能地按照元素在内存中的顺序访问，利用高速缓存达到加速的目的。因此，在选择行优先和列优先时，应当充分考虑可能的操作。例如，矩阵与向量的乘积至少可以有如下两种实现方式。

算法 28 矩阵与向量的乘积（方式1）

1: **for** $j \in \{1, 2, \ldots, n\}$ **do**
2: $b_j \leftarrow 0$。
3: **end for**
4: **for** $i \in \{1, 2, \ldots, m\}$ **do**
5: **for** $j \in \{1, 2, \ldots, n\}$ **do**
6: $b_j \leftarrow b_j + a_{i,j} x_j$。
7: **end for**
8: **end for**

算法 29 矩阵与向量的乘积（方式2）

1: **for** $j \in \{1, 2, \ldots, n\}$ **do**
2: $b_j \leftarrow 0$。
3: **for** $i \in \{1, 2, \ldots, m\}$ **do**
4: $b_j \leftarrow b_j + a_{i,j} x_j$。
5: **end for**
6: **end for**

算法28和算法29给出的这两种实现方式在本质上是相同的，主要是循环顺序的差别。如果矩阵是按照行优先顺序存储的，那么通常算法28的速度更快。相反，如果矩阵是按照列优先顺序存储的，那么通常算法29的速度更快。这是因为矩阵向量乘积恰好访问矩阵中的每个元素各一次，如果能按照矩阵元素在内存中的顺序来访问，那么计算速度会更快。对于矩阵乘积也有类似的问题，但会更为复杂。因为矩阵乘积会涉及两个矩阵，并且会多次访问矩阵中的每个元素。在设计算法之前，选择行优先还是列优先存储是非常重要的一步。

如果不愿意自己考虑具体实现上的差别，则可以考虑借助 BLAS（Basic Linear Algebra Subprograms）。BLAS 是矩阵向量运算最常用的接口规范，目前已经有许多流行的函数库支持这个接口规范，例如 Intel MKL、ATLAS、OpenBLAS 等。自己编写一个非常基本的矩阵运算程序通常很难超越这些函数库，这是因为函数库在编写时已经充分考虑了计算机缓存等非常细节的效率问题。BLAS 将矩阵向量运算分为三个级别，划分的依据主要是时间复杂度。第 1 级的时间复杂度是线性，第 2 级的时间复杂度是二次，第 3 级的时间复杂度是三次。由于第 1 级的运算非常简单，可优化的空间不大，所以通常自己编写也能达到类似的计算效率。相对而言，第 2 级和第 3 级，尤其是第 3 级的优化空间较大，使用函数库的优势会更为明显。许多语言都有工具包为 BLAS 包装了更为高级的接口，例如 Python 语言中的 NumPy 包。

4.3 三角方程组

求解三角方程组实质上是高斯消去法中的重要一步。因此在研究一般方程组之前，我们先来介绍三角方程组。

4.3.1 三角矩阵

三角矩阵（Triangular Matrix）是方阵中的一类，是下三角矩阵和上三角矩阵的统称。一个方阵，如果其对角线上方的元素全是 0，则叫作下三角矩阵；如果其对角线下方的元素全是 0，则叫作上三角矩阵。对角矩阵既是下三角矩阵又是上三角矩阵。

- 单位三角矩阵（Unitriangular Matrix）：对角线上的元素全等于 1 的三角矩阵。例如

$$\begin{pmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ -2 & 8 & 1 \end{pmatrix} \tag{4.2}$$

- 严格三角矩阵（Strictly Triangular Matrix）：对角线上的元素全等于 0 的三角矩阵。例如

$$\begin{pmatrix} 0 & 0 & 0 \\ 3 & 0 & 0 \\ -2 & 8 & 0 \end{pmatrix} \tag{4.3}$$

- 高斯变换矩阵（Gaussian Transformation Matrix），或称 Frobenius 矩阵：除了某一列的对角线下方，非对角线上全是 0 的单位三角矩阵。

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -2 & 1 & 0 \\ 0 & 4 & 0 & 1 \end{pmatrix} \tag{4.4}$$

三角矩阵有很多非常好的性质。以上三角矩阵为例，我们有：

- 两个上三角矩阵的和是上三角矩阵；
- 两个上三角矩阵的乘积是上三角矩阵；
- 上三角矩阵的逆是上三角矩阵。

这样的性质对于下三角矩阵也相应成立。

4.3.2 三角矩阵的存储

三角矩阵可以按照普通矩阵存储，比如前面所介绍的行优先或列优先方式。这样做的优点是编程逻辑简单，缺点是会浪费一半左右的空间去存储 0。一种解决方式就是仅存储对角线及其一侧的数。例如

$$\begin{pmatrix} 1 & 0 & 0 \\ 3 & 6 & 0 \\ -2 & 0 & 5 \end{pmatrix} \tag{4.5}$$

可以存储为 1, 3, 6, -2, 0, 5（行优先）或 1, 3, -2, 6, 0, 5（列优先）。这样都可以节约大致一半的存储空间。但在查找指定位置的元素时，在逻辑上会稍显复杂。可以验证，对于一个 $n \times n$ 的下三角矩阵，$a_{i,j}$ 的存储位置在行优先时是 $i(i-1)/2+j$，在列优先时是 $(2n-j+2)(j-1)/2+i$。因此这个存储方式在节约空间的同时带来了一些多余的计算。

如果需要同时存储多个三角矩阵，那么还有一种有意思的方式是，将两个三角矩阵拼在一起存成一个完整的一般矩阵。唯一的问题是对角线位置仅能存储其中一个三角矩阵的对角线，而另外一个三角矩阵的对角线需要额外的存储空间。如果其中一个三角矩阵是单位三角矩阵或者严格三角矩阵，则其对角线不必单独存储，在和其他三角矩阵拼起来存储时会更为方便。类似地，我们也可以将多个高斯变换矩阵拼在一起存储。

4.3.3 三角方程组求解

$\boldsymbol{Lx} = \boldsymbol{b}$ 和 $\boldsymbol{Ux} = \boldsymbol{b}$ 都是非常容易求解的。我们一般使用向前消去法（Forward Substitution）求解 $\boldsymbol{Lx} = \boldsymbol{b}$。

我们将 $\boldsymbol{Lx} = \boldsymbol{b}$ 展开如下。

$$\begin{cases} L_{1,1}x_1 & & & & & = & b_1 \\ L_{2,1}x_1 & + & L_{2,2}x_2 & & & = & b_2 \\ \vdots & & \vdots & & & \vdots & \vdots \\ L_{n,1}x_1 & + & L_{n,2}x_2 & + \cdots + & L_{n,n}x_n & = & b_n \end{cases} \tag{4.6}$$

向前消去法的思路就是按顺序求出 $x_1, x_2, \ldots, x_n$。可以将已求出的变量值代入其他方程，从而消去这个变量。具体来说，在第 1 个方程中仅包含一个变量 x_1，因此我们可以直接求出 x_1。将 x_1 的值代入到第 2 个方程后，在第 2 个方程中就仅剩一个变量 x_2 了，因此我们可以直接求出 x_2。再将 x_1, x_2 的值代入第 3 个方程，在第 3 个方程中就仅剩一个变量 x_3 了，因此我们可以直接求出 x_3。以此类推。

最终我们可以将计算结果写成如下形式。

$$
\begin{cases}
x_1 &= \dfrac{b_1}{L_{1,1}} \\
x_2 &= \dfrac{b_2 - L_{2,1}x_1}{L_{2,2}} \\
&\vdots \\
x_n &= \dfrac{b_n - \sum\limits_{i=1}^{n-1} L_{n,i}x_i}{L_{n,n}}
\end{cases} \tag{4.7}
$$

在这个过程中，我们总共需要计算的加减法次数为 $0+1+\cdots+(n-1)=\dfrac{n(n-1)}{2}$，乘法次数为 $0+1+\cdots+(n-1)=\dfrac{n(n-1)}{2}$，除法次数为 n。因此，这一算法的时间复杂度为 $O(n^2)$。注意到，即使遍历一次下三角矩阵中的所有元素，也需要 $O(n^2)$ 的时间，所以我们可以看到向前消去法的计算速度是非常快的。

类似地，我们可以用向后消去法（Backward Substitution）求解 $\boldsymbol{U}\boldsymbol{x}=\boldsymbol{b}$。

我们将 $\boldsymbol{U}\boldsymbol{x}=\boldsymbol{b}$ 展开如下。

$$
\begin{cases}
U_{1,1}x_1 + U_{1,2}x_2 + \cdots + U_{1,n}x_n = b_1 \\
\qquad\quad U_{2,2}x_2 + \cdots + U_{2,n}x_n = b_2 \\
\qquad\qquad\qquad\qquad \vdots \quad \vdots \\
\qquad\qquad\qquad\quad U_{n,n}x_n = b_n
\end{cases} \tag{4.8}
$$

与向前消去法的区别是，我们要从最后一个方程出发，依次向前求出 $x_n, x_{n-1}, \cdots, x_1$。计算结果如下。

$$
\begin{cases}
x_n &= \dfrac{b_n}{U_{n,n}} \\
x_{n-1} &= \dfrac{b_{n-1} - U_{n-1,n}x_n}{U_{n-1,n-1}} \\
&\vdots \\
x_1 &= \dfrac{b_1 - \sum\limits_{i=2}^{n} U_{1,i}}{U_{1,1}}
\end{cases} \tag{4.9}
$$

同向前消去法一样，向后消去法的时间复杂度也是 $O(n^2)$。

4.4 高斯消去法

4.4.1 算法概述

高斯消去法是一种直接法，它直接在方程上进行操作。过程大致是：反复将某一行减去“另一行的某个倍数”，最终将原方程组变为上三角方程组。因为两个等式相减仍然是等式，所以整个算法过程始终是等价变换。我们来看这样一个具体的方程组：

$$\begin{aligned} x_1 &+ 2x_2 &+ 3x_3 &= 1 \\ 2x_1 &+ 6x_2 & &= 1 \\ x_1 & & + 7x_3 &= 1 \end{aligned} \tag{4.10}$$

为了书写方便，我们将其写为如下增广矩阵的形式。

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 2 & 6 & 0 & 1 \\ 1 & 0 & 7 & 1 \end{array}\right) \tag{4.11}$$

在使用高斯消去法时，我们的目的是将原方程组变为上三角方程组。因此在第 1 列中，第 2 行和第 3 行都应当是 0。为了将它们变为 0，我们让第 2 行减去第 1 行的 2 倍，让第 3 行减去第 1 行的 1 倍。

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 2 & 6 & 0 & 1 \\ 1 & 0 & 7 & 1 \end{array}\right) \to \left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & 2 & -6 & -1 \\ 0 & -2 & 4 & 0 \end{array}\right) \tag{4.12}$$

现在距离上三角方程组已经很近了，只剩下第 2 列中第 3 行应当是 0。我们让第 3 行减去第 2 行的 -1 倍。

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & 2 & -6 & -1 \\ 0 & -2 & 4 & 0 \end{array}\right) \to \left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & 2 & -6 & -1 \\ 0 & 0 & -2 & -1 \end{array}\right) \tag{4.13}$$

因此，原方程组等价于

$$\begin{aligned} x_1 &+ 2x_2 &+ 3x_3 &= 1 \\ & 2x_2 &- 6x_3 &= -1 \\ & &- 2x_3 &= -1 \end{aligned} \tag{4.14}$$

这时，我们得到了一个上三角方程组。于是我们可以使用前面介绍过的方法求解这一三角方程组的解。因此高斯消去法的大致流程可以写成算法30。

算法 30 高斯消去法

1: **for** $i \in \{1, 2, \ldots, n\}$ **do**
2: **for** $j \in \{i+1, i+2, \ldots, n\}$ **do**
3: 让第 j 行减去第 i 行的若干倍，从而将第 j 行第 i 列变为 0。
4: **end for**
5: **end for**
6: 求解上三角方程组。

高斯消去法有一个特点。在第 1 轮消去后，第 1 行和第 1 列就不需要再考虑了。在第 2 轮消去后，第 2 行和第 2 列也不需要再考虑了。因此，在第 i 轮消去后，我们称第 $i+1$ 行至第 n 行及第 $i+1$ 列至第 n 列所构成的子矩阵为活跃子矩阵（Active Sub-matrix）。在每一轮消去的时候，我们不需要考虑整个矩阵，只需考虑活跃子矩阵。

然而算法30还存在很多细节问题。其中最大的问题是，如果在某一步无法将第 j 行第 i 列变为 0，则该如何处理。这种情况一定是因为第 i 行第 i 列为 0 导致的，因为我们实际上是在利用第 i 行第 i 列将对角线下方的元素变为 0，这个重要的元素也被称为主元（Pivot Element）。如果主元为 0，我们就需要更换其他元素来做主元。这时我们可以调整行的顺序，把第 $i+1, i+2, \ldots, n$ 行中的某一行变为第 i 行。调整方程顺序并不影响方程组的解，因此这样做是可行的。这样的一个调整顺序的过程叫作选主元。在后面我们还会详细地介绍选主元的方法。

另外一个极端情况是第 $i, i+1, i+2, \ldots, n$ 行的第 i 列都是 0，这种情况说明第 i 个变量是冗余的，可以随意取值。这种情况会同时伴随行冗余，可能导致方程组无解。

4.4.2 高斯变换

高斯消去法的每一步可以看成做一次高斯变换。我们仍然通过举例来说明高斯消去法和高斯变换的关系。考虑使用高斯消去法求解如下方程。

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 6 & 0 \\ 1 & 0 & 7 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \tag{4.15}$$

我们可以使用矩阵乘法来描述高斯消去的过程。同之前一样，我们首先让第 2 行减去第 1 行的 2 倍，让第 3 行减去第 1 行的 1 倍。这一步可以看成做一次高斯变换。

$$\boldsymbol{L}_1\boldsymbol{A} = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 2 & 6 & 0 \\ 1 & 0 & 7 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 2 & -6 \\ 0 & -2 & 4 \end{pmatrix} \tag{4.16}$$

然后，我们让第 3 行减去第 2 行的 -1 倍。这一步同样可以看成做一次高斯变换。

$$\boldsymbol{L}_2\boldsymbol{L}_1\boldsymbol{A} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 0 & 2 & -6 \\ 0 & -2 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 2 & -6 \\ 0 & 0 & -2 \end{pmatrix} \tag{4.17}$$

最后我们再来了解高斯变换矩阵的一个有趣性质。我们知道下三角矩阵的逆矩阵是下三角矩阵，而高斯变换矩阵也是下三角矩阵，所以高斯变换矩阵的逆矩阵是下三角矩阵。实际上还有更强的结论：高斯变换矩阵的逆矩阵是高斯变换矩阵。我们以之前的两个高斯变换矩阵为例：

$$\boldsymbol{L}_1 = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix}, \boldsymbol{L}_1^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \tag{4.18}$$

$$\boldsymbol{L}_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}, \boldsymbol{L}_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix} \tag{4.19}$$

可能读者已经看出高斯变换矩阵与其逆矩阵的规律。我们有如下定理。

定理 4.1 高斯变换矩阵 $\boldsymbol{L}$ 的逆矩阵是 $2\boldsymbol{I}-\boldsymbol{L}$，仍然是一个高斯变换矩阵。其中 $\boldsymbol{I}$ 是单位矩阵。

4.4.3 LU 分解

提到高斯消去法，就必须提到 LU 分解。实际上 LU 分解基本上是和高斯消去法在做同样一件事情。LU 分解的 L 表示 Lower Triangular Matrix，即下三角矩阵；U 表示 Upper Triangular Matrix，即上三角矩阵。这个分解不是唯一的，但通常我们会让 L 成为单位下三角矩阵。我们先来看之前的矩阵做 LU 分解后是个什么结果，然后得到其中的联系。

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 6 & 0 \\ 1 & 0 & 7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 0 & 2 & -6 \\ 0 & 0 & -2 \end{pmatrix} \tag{4.20}$$

其中

$$\boldsymbol{L} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & -1 & 1 \end{pmatrix}; \boldsymbol{U} = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 2 & -6 \\ 0 & 0 & -2 \end{pmatrix} \tag{4.21}$$

联系之前的高斯消去法可以发现，这里的 $\boldsymbol{U}$ 就是在高斯消去法中得到的上三角矩阵，而这里的 $\boldsymbol{L}$ 也与前面提到的高斯变换矩阵非常接近。

通过进一步观察可以发现，如果矩阵 A 在高斯消去法后得到

$$\boldsymbol{L}_n\boldsymbol{L}_{n-1}\cdots\boldsymbol{L}_2\boldsymbol{L}_1\boldsymbol{A} = \boldsymbol{U} \tag{4.22}$$

那么我们就可以得到矩阵 A 的一个分解：

$$\boldsymbol{A} = \boldsymbol{L}_1^{-1}\boldsymbol{L}_2^{-1}\cdots\boldsymbol{L}_{n-1}^{-1}\boldsymbol{L}_n^{-1}\boldsymbol{U} \tag{4.23}$$

下三角矩阵的逆是下三角矩阵，下三角矩阵的乘积是下三角矩阵。所以可以记 $\boldsymbol{L} = \boldsymbol{L}_1^{-1}\boldsymbol{L}_2^{-1}\cdots\boldsymbol{L}_{n-1}^{-1}\boldsymbol{L}_n^{-1}$，这样就有 $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}$。这正好就是 LU 分解。前面已经提到，我们可以直接写出高斯变换的逆矩阵，所以有

$$\boldsymbol{A} = (2\boldsymbol{I}-\boldsymbol{L}_1)(2\boldsymbol{I}-\boldsymbol{L}_2)\cdots(2\boldsymbol{I}-\boldsymbol{L}_n)\boldsymbol{U}$$

高斯变换的逆矩阵仍然是高斯变换矩阵，也就是说 $2\boldsymbol{I}-\boldsymbol{L}_1$、$2\boldsymbol{I}-\boldsymbol{L}_2\cdots\cdots 2\boldsymbol{I}-\boldsymbol{L}_n$ 都是高斯变换矩阵。更有意思的是，这 n 个矩阵的乘积是不需要计算的，可以直接写出。可以验证，$\boldsymbol{L}$ 的对角线元素全是 1，对角线上方全是 0，对角线下方的第 1 列与 $2\boldsymbol{I}-\boldsymbol{L}_1$ 相同，对角线下方的第 2 列与 $2\boldsymbol{I}-\boldsymbol{L}_2$ 相同……对角线下方的第 n 列与 $2\boldsymbol{I}-\boldsymbol{L}_n$ 相同。因此在高斯消去法后，我们只需简单处理就可以得到 LU 分解。

虽然计算过程基本是一致的，但高斯消去法主要是指求解线性方程组，而 LU 分解则有更广泛的用途。

- LU 分解可以用于求逆矩阵。我们记 $\boldsymbol{e}_i$ 为第 i 行是 1 且其余行都是 0 的列向量。因为逆矩阵 $\boldsymbol{A}^{-1}$ 满足 $\boldsymbol{A}\boldsymbol{A}^{-1}=\boldsymbol{I}$，所以 $\boldsymbol{A}^{-1}$ 的第 i 列就是方程组 $\boldsymbol{A}\boldsymbol{x}=\boldsymbol{e}_i$ 的解。我们只需对 $\boldsymbol{A}$ 计算一次 LU 分解，再求解 n 次下三角方程组、n 次上三角方程组，就可以得到 $\boldsymbol{A}$ 的逆矩阵了。
- LU 分解还可以用于求行列式。因为 $\det(\boldsymbol{A})=\det(\boldsymbol{L}\boldsymbol{U})=\det(\boldsymbol{L})\det(\boldsymbol{U})$，而三角矩阵的行列式就是对角线元素的乘积，所以 $\det(\boldsymbol{A})$ 就等于 $\boldsymbol{U}$ 的对角线元素的乘积。这样在 LU 分解后，我们就可以很快得到行列式了。

4.4.4 Cholesky 分解

如果 $\boldsymbol{L}$ 是一个下三角矩阵，那么 $\boldsymbol{L}^{\mathrm{T}}$ 就是上三角矩阵，所以一个非常特殊的 LU 分解就是 $\boldsymbol{U}=\boldsymbol{L}^{\mathrm{T}}$，即将矩阵分解为 $\boldsymbol{L}\boldsymbol{L}^{\mathrm{T}}$。这种分解也叫作 Cholesky 分解。并不是所有矩阵都存在这样的分解，但如果矩阵是半正定矩阵，我们就可以这样分解。事实上，一个矩阵当且仅当它可以分解为 $\boldsymbol{L}\boldsymbol{L}^{\mathrm{T}}$ 时是半正定矩阵，其中 $\boldsymbol{L}$ 是下三角矩阵。

我们记半正定矩阵 $\boldsymbol{A}=\boldsymbol{L}\boldsymbol{L}^{\mathrm{T}}$，则直接根据矩阵乘法的规则，有 $\boldsymbol{A}_{1,1}=\boldsymbol{L}_{1,1}^2$，因此可以直接算出 $\boldsymbol{L}_{1,1}=\sqrt{\boldsymbol{A}_{1,1}}$。类似地，有 $\boldsymbol{A}_{1,2}=\boldsymbol{L}_{1,1}\boldsymbol{L}_{2,1}$，因此可以直接算出 $\boldsymbol{L}_{2,1}=\dfrac{\boldsymbol{A}_{1,2}}{\boldsymbol{L}_{1,1}}$。以此类推，我们可以得到 Cholesky 分解的算法流程如下。

算法 31 Cholesky 分解

1: **for** $i=1,2,\cdots,n$ **do**
2: **for** $j=1,2,\cdots,i-1$ **do**
3: **if** $L_{j,j}\neq 0$ **then**
4: $\boldsymbol{L}_{i,j}\leftarrow\dfrac{\boldsymbol{A}_{i,j}-\sum\limits_{k=1}^{j-1}\boldsymbol{L}_{i,k}\boldsymbol{L}_{j,k}}{\boldsymbol{L}_{j,j}}$。
5: **else**

6: $\quad\quad \boldsymbol{L}_{i,j} \leftarrow 0$。

7: $\quad$ **end if**

8: **end for**

9: $\boldsymbol{L}_{i,i} \leftarrow \sqrt{\boldsymbol{A}_{i,i} - \sum_{j=1}^{i-1} \boldsymbol{L}_{i,j}^2}$。

10: **end for**

有一点值得注意的是，在算法中需要避免分母为 0 的情况。因此在算法31的第4行中，只有在 $\boldsymbol{L}_{j,j} \neq 0$ 的时候我们才会进行相应的计算，而在 $\boldsymbol{L}_{j,j} = 0$ 的时候，我们可以利用半正定矩阵的性质证明 $\boldsymbol{L}_{i,j} = 0$。在实际计算中，如果 $\boldsymbol{L}_{j,j}$ 非常小，则我们也认为它是 0。

我们可以看到，Cholesky 分解虽然实质上也是在做高斯消去法，但由于半正定矩阵的特殊性，其过程要简洁很多。

4.5 主元选择

4.5.1 列选主元

我们考虑这样一个线性方程组

$$\begin{cases} \epsilon \quad x_1 \quad + \quad x_2 \quad = \quad 1 \\ \quad\quad x_1 \quad + \quad x_2 \quad = \quad 2 \end{cases} \tag{4.24}$$

其中，ϵ 是一个非常小的正实数。我们可以手算得到这个方程的解：

$$\begin{cases} x_1 \quad = \quad \dfrac{1}{1-\epsilon} \quad \approx \quad 1 \\ x_2 \quad = \quad \dfrac{1-2\epsilon}{1-\epsilon} \quad \approx \quad 1 \end{cases} \tag{4.25}$$

在计算机中，由于精度原因，计算结果会出现同手算不一致的情况。此时高斯消去法的过程可能是这样的：

$$\left(\begin{array}{cc|c} \epsilon & 1 & 1 \\ 1 & 1 & 2 \end{array}\right) \rightarrow \left(\begin{array}{cc|c} \epsilon & 1 & 1 \\ 0 & 1-1/\epsilon & 2-1/\epsilon \end{array}\right) \tag{4.26}$$

使用向后消去法，我们得到

$$\begin{cases} x_2 &= \dfrac{2-1/\epsilon}{1-1/\epsilon} \\ x_1 &= \dfrac{1-x_2}{\epsilon} \end{cases} \tag{4.27}$$

这个表达式乍一看没有什么问题，但在实际计算时可能会有意外发生。在笔者的计算机上，5 个不同的 ϵ 取值就会产生不一致的计算结果。我们将结果整理为表4.1。可以看到，在 ϵ 非常小的时候，甚至出现了 $x_1 = 0, x_2 = 1$ 的情况，而这个解完全不满足方程组。

表 4.1 式(4.27)不同 ϵ 对应的计算结果

ϵ	x_1	x_2
10^{-8}	1.000000	1.000000
10^{-12}	0.999978	1.000000
10^{-16}	2.220446	1.000000
10^{-20}	0.000000	1.000000
10^{-24}	0.000000	1.000000

如果我们交换两个方程的顺序，再应用高斯消去法，则结果会有很大的不同。

$$\left(\begin{array}{cc|c} 1 & 1 & 2 \\ \epsilon & 1 & 1 \end{array}\right) \rightarrow \left(\begin{array}{cc|c} 1 & 1 & 2 \\ 0 & 1-\epsilon & 1-2\epsilon \end{array}\right) \tag{4.28}$$

使用向后消去法，我们得到

$$\begin{cases} x_2 &= \dfrac{1-2\epsilon}{1-\epsilon} \\ x_1 &= 2-x_2 \end{cases} \tag{4.29}$$

这时再次采用不同的 ϵ 进行测试，计算结果如表 4.2所示。

表 4.2 式(4.29)中不同 ϵ 对应的计算结果

ϵ	x_1	x_2
10^{-8}	1.000000	1.000000
10^{-12}	1.000000	1.000000
10^{-16}	1.000000	1.000000
10^{-20}	1.000000	1.000000
10^{-24}	1.000000	1.000000

这就说明，调整行的顺序虽然在逻辑上不影响解，但在实际计算中会影响误差。不谨慎地进行顺序调整，可能会把误差放大成很严重的错误。本节所用的调整方法就是列选主元。所谓列选主元也就是通过调整行顺序，使得当前列上的主元的绝对值尽量大，这样通常可以减小误差。

列选主元主要适用于一般的 LU 分解，而对于 Cholesky 分解是不可以使用列选主元的。这是因为单纯调整行的顺序会影响矩阵的对称性，从而影响了半正定的性质。

4.5.2 全选主元

全选主元不但考虑调整行的顺序，还考虑调整列的顺序，使得主元的绝对值尽量大。下面一个例子说明仅仅调整行可能是不够的。在这个例子中，第 1 列的两个元素相同，因此不需要调整行顺序。

$$\left(\begin{array}{cc|c} 1 & 1/\epsilon & 1/\epsilon \\ 1 & 1 & 2 \end{array}\right) \rightarrow \left(\begin{array}{cc|c} 1 & 1/\epsilon & 1/\epsilon \\ 0 & 1-1/\epsilon & 2-1/\epsilon \end{array}\right) \tag{4.30}$$

使用向后消去法，我们得到

$$\begin{cases} x_2 = \dfrac{2-1/\epsilon}{1-1/\epsilon} \\ x_1 = 1/\epsilon - 1/\epsilon \cdot x_2 \end{cases} \tag{4.31}$$

与之前类似，我们采用不同的 ϵ 进行测试，结果如表 4.3所示。

表 4.3 式(4.31)中不同 ϵ 对应的计算结果

ϵ	x_1	x_2
10^{-8}	1.000000	1.000000
10^{-12}	1.000000	1.000000
10^{-16}	2.000000	1.000000
10^{-20}	0.000000	1.000000
10^{-24}	0.000000	1.000000

这次我们又看到了明显错误的解，而列选主元并不能解决这次的问题。如果采用全选主元，则由于 $1/\epsilon$ 是最大的元素，我们将它作为主元重新计算。注意到这时我们调整了列的顺序，也就是交换了 x_1 和 x_2 的顺序。在最终的计算结果中，应当注意不要搞混 x_1 和 x_2

的取值，它们应当与原始方程组的顺序一致。

$$\left(\begin{array}{cc|c} 1/\epsilon & 1 & 1/\epsilon \\ 1 & 1 & 2 \end{array}\right) \rightarrow \left(\begin{array}{cc|c} 1/\epsilon & 1 & 1/\epsilon \\ 0 & 1-\epsilon & 1 \end{array}\right) \tag{4.32}$$

使用向后消去法，我们得到

$$\begin{cases} x_1 &= 1/(1-\epsilon) \\ x_2 &= \dfrac{1/\epsilon - x_1}{1/\epsilon} \end{cases} \tag{4.33}$$

再次采用不同的 ϵ 进行测试，结果如表 4.4所示。

表 4.4 式(4.33)不同 ϵ 对应的计算结果

ϵ	x_1	x_2
10^{-8}	1.000000	1.000000
10^{-12}	1.000000	1.000000
10^{-16}	1.000000	1.000000
10^{-20}	1.000000	1.000000
10^{-24}	1.000000	1.000000

这说明，在某些情况下，全选主元确实可以弥补列选主元的不足。但使用全选主元的代价太大，每次选主元的时候都要对活跃子矩阵进行遍历，时间成本太高。因此除了列选主元和全选主元，还有一种折中的选主元方法：部分选主元。部分选主元选择性地考虑一部分行和列来调整主元。

另外，选主元在用于 Cholesky 分解的时候，为了保持矩阵的对称性，我们只能选择对角线元素作为主元。

4.5.3 主元与计算量

主元除了影响数值的稳定性，还会影响计算量。我们再次以前面的方程组为例。在不选主元的情况下，消去过程是这样的：

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 6 & 0 \\ 1 & 0 & 7 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 2 & 3 \\ 0 & 2 & -6 \\ 0 & -2 & 4 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 2 & 3 \\ 0 & 2 & -6 \\ 0 & 0 & -2 \end{pmatrix}$$

注意到这两轮消去分别改变了矩阵中的 6 个元素和两个元素。但如果我们将方程的顺序进行交换，则高斯消去会变为另外一个过程：

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 6 & 0 \\ 1 & 0 & 7 \end{pmatrix} \to \begin{pmatrix} 1 & 0 & 7 \\ 1 & 2 & 3 \\ 2 & 6 & 0 \end{pmatrix} \to \begin{pmatrix} 1 & 0 & 7 \\ 0 & 2 & -4 \\ 0 & 6 & -14 \end{pmatrix} \to \begin{pmatrix} 1 & 0 & 7 \\ 0 & 2 & -4 \\ 0 & 0 & -2 \end{pmatrix}$$

其中的第 1 步，我们仅仅改变了行顺序，后两步为高斯消去。与之前不同，这两轮消去分别改变了矩阵中的 4 个元素和 2 个元素，也就是说我们节约了一些计算量。这个节约就来自于矩阵中的零元素，因为零元素实质上并不影响计算结果，所以也就不用参与计算。对于大规模问题，矩阵通常是稀疏的，也就是说有很多个零元素。这时与数值稳定性相比，选主元的更大作用是减少计算量，其影响可能是巨大的。我们考虑这样一类特殊的矩阵，其对角线上全是 3，第 1 行及第 1 列上除对角线外全是 1，其余位置全是 0。以 5 阶方阵为例，可以写成

$$\begin{pmatrix} 3 & 1 & 1 & 1 & 1 \\ 1 & 3 & 0 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 \\ 1 & 0 & 0 & 3 & 0 \\ 1 & 0 & 0 & 0 & 3 \end{pmatrix} \tag{4.34}$$

如果在高斯消去法的第 1 轮消去中选择第 1 行第 1 列作为主元，则该矩阵的每个位置都需要参与消去的计算。但如果高斯消去法依次选择第 5 行第 5 列、第 4 行第 4 列……第 2 行第 2 列作为主元，则可以验证，在全部计算结束后，该矩阵也仅有非零元素参与消去的计算。而非零元素与全部元素相比完全不在一个量级。

在后面的几节中，我们将着重考虑稀疏矩阵，探讨如何在高斯消去法或者 LU 分解中节约计算量。

4.6 稀疏矩阵的编程基础

稀疏矩阵是指大部分元素是 0 的一类矩阵。反之，如果大部分元素不是 0，则叫作稠密矩阵。非零元素的个数与矩阵元素总个数的比值叫作稀疏度。在大规模问题中，我们遇到的矩阵通常都是稀疏矩阵，虽然矩阵本应很大，但是其中的非零元素并不多。如果仍然按

照稠密矩阵的策略去进行处理，就会非常吃亏。因为零元素常常不影响计算结果，所以没有必要每次都对它们进行处理。

稀疏矩阵的存储相对于稠密矩阵的存储更有技巧性。由于零元素非常多，所以我们仅需记录非零元素的信息，这时就很难同时使常见操作都很快速。一些数据结构适合矩阵修改操作，我们称其为适合动态存储的数据结构。而另外一些数据结构适合矩阵遍历操作，我们称其为适合静态存储的数据结构。对于一个具体的问题，应当根据需求来选择最为合适的数据结构。对于一些复杂的情况，一个数据结构往往不足以解决问题。例如在某些场景下，我们刚开始需要对矩阵进行添加元素等修改，而之后则主要用其参与运算，基本不再修改。这时就需要先构建一个适合动态存储的数据结构，待矩阵不再需要修改时，将其转化为适合静态存储的数据结构。

4.6.1 稀疏向量

稀疏向量是稀疏矩阵的基础，因此我们先从稀疏向量下手。不同于稠密向量，稀疏向量的基本运算是非常有技巧性的。既然我们认为稀疏向量中的元素几乎以零为主，我们就应当只存储非零元素。例如一个 10 维的向量 $(2,0,0,0,5,1,0,0,0,0)$ 仅包含 3 个非零元素，则我们可以存储这样一些信息。

- 维数，n：10
- 非零元个数，length：3
- 非零元位置，index：$1,5,6$
- 非零元数值，value：$2,5,1$

以上这些信息足以唯一确定我们所需要的系数向量，并且存储空间要小于稠密向量。请注意这里非零元位置和非零元数值的存储位置需要有正确的对应关系。这里还有一个细节，就是非零元的存储顺序问题。在上述例子中，非零元是按照位置排序的，但即使不排序，只要非零元位置和数值是对应的，则同样可以唯一确定一个稀疏向量，例如

- 维数，n：10
- 非零元个数，length：3
- 非零元位置，index：$5,6,1$
- 非零元数值，value：$5,1,2$

也表述了完全相同的一个稀疏向量。

是否按顺序存储非零元各有利弊，需要根据实际情况确定。我们以稀疏向量加法为例，考虑稀疏向量 $\boldsymbol{u}$ 和稀疏向量 $\boldsymbol{v}$，我们要计算 $\boldsymbol{u} \leftarrow \boldsymbol{u} + \boldsymbol{v}$。

如果非零元的位置是按顺序存储的，则我们可以同时顺序扫描 $\boldsymbol{u}$ 和 $\boldsymbol{v}$，这一点确实非常方便。但是如果存在某个位置 i 使得 $v_i \neq 0$，但 $u_i = 0$，那么就需要在 $\boldsymbol{u}$ 中插入一个新的非零元。这时如果要继续维护 $\boldsymbol{u}$ 中元素的顺序，就必须在其中插入新的元素。因此，非零元位置和非零元数值需要以链表形式存储来降低插入操作的时间复杂度。完整的算法流程如下。

算法 32 稀疏向量加法（非零元有序）

$i \leftarrow 1$，$j \leftarrow 1$。
while $i \leqslant \text{length}_u$ 且 $j \leqslant \text{length}_v$ **do**
  if $\text{index}_u(i) < \text{index}_v(j)$ **then**
    $i \leftarrow i + 1$。
  else if $\text{index}_u(i) > \text{index}_v(j)$ **then**
    在 index_u 的第 i 个位置插入 $\text{index}_v(j)$。
    在 value_u 的第 i 个位置插入 $\text{value}_v(j)$。
    $i \leftarrow i + 1$，$j \leftarrow j + 1$。
  else
    $\text{value}_u(i) \leftarrow \text{value}_u(i) + \text{value}_v(j)$。
    $i \leftarrow i + 1$，$j \leftarrow j + 1$。
  end if
end while
while $j \leqslant \text{length}_v$ **do**
  在 index_u 的末尾插入 $\text{index}_v(j)$。
  在 value_u 的末尾插入 $\text{value}_v(j)$。
  $j \leftarrow j + 1$。
end while

如果非零元的位置不必按顺序存储，则我们可以将其中一个稀疏向量临时转换为稠密向量，再进行计算。我们假设有一个长为 n 的辅助存储空间 $\boldsymbol{w}$，且已事先赋值为 0，则算法流程如下。

算法 33 稀疏向量加法（非零元无序）

1: **for** $i = 1, 2, \ldots,$ length$_v$ **do**
2: $\quad w(\text{index}_v(i)) \leftarrow \text{value}_v(i)$。
3: **end for**
4: **for** $i = 1, 2, \ldots,$ length$_u$ **do**
5: $\quad$ **if** $w(\text{index}_u(i)) \neq 0$ **then**
6: $\quad\quad \text{value}_u(i) \leftarrow \text{value}_u(i) + w(\text{index}_u(i))$。
7: $\quad\quad w(\text{index}_u(i)) \leftarrow 0$。
8: $\quad$ **end if**
9: **end for**
10: **for** $j = 1, 2, \ldots,$ length$_v$ **do**
11: $\quad$ **if** $w(\text{index}_v(i)) \neq 0$ **then**
12: $\quad\quad \text{length}_u \leftarrow \text{length}_u + 1$。
13: $\quad\quad \text{index}_u(\text{length}_u) \leftarrow \text{index}_v(i)$。
14: $\quad\quad \text{value}_u(\text{length}_u) \leftarrow \text{value}_v(i)$。
15: $\quad\quad w(\text{index}_v(i)) \leftarrow 0$。
16: $\quad$ **end if**
17: **end for**

注意这里我们最后将向量 $\boldsymbol{w}$ 恢复为零向量，这样可以便于下次使用。如果在每次需要计算稀疏向量加法时都先将稠密向量 $\boldsymbol{w}$ 赋值为 0，那么由于 $\boldsymbol{w}$ 是稠密存储，则时间复杂度将大大增加。与此相比，如果我们每次在使用零向量 $\boldsymbol{w}$ 后，将其恢复为零向量，则这样在下次使用时就不必初始化了。而将 $\boldsymbol{w}$ 恢复为零向量的代价是非常低的。可能有读者好奇为什么不将非零元排序后再处理，这是因为排序会导致时间复杂度更高。

另外一个常见的计算就是稀疏向量内积。稀疏向量内积比求和稍简单一些，但思路非常接近。如果非零元有序，则只需顺序扫描。

算法 34 稀疏向量内积（非零元有序）

1: $i \leftarrow 1$，$j \leftarrow 1$。
2: **while** $i \leqslant \text{length}_u$ 且 $j \leqslant \text{length}_v$ **do**
3: $\quad$ **if** $\text{index}_u(i) < \text{index}_v(j)$ **then**
4: $\quad\quad i \leftarrow i + 1$。

```
    else if index_u(i) > index_v(j) then
        j ← j + 1。
    else
        r ← r + value_u(i) × value_v(i)。
    end if
end while
```

如果非零元无序，则我们也是先将稀疏向量 $\boldsymbol{v}$ 转化为稠密存储的向量 $\boldsymbol{w}$，再进行计算。

算法 35 稀疏向量内积（非零元无序）

```
for i = 1, 2, ..., length_v do
    w(index_v(i)) ← value_v(i)。
end for
r ← 0。
for i = 1, 2, ..., length_v do
    r ← r + w(index_v(i)) × value_v(i)。
end for
for i = 1, 2, ..., length_v do
    w(index_v(i)) ← 0。
end for
```

这里需要注意的是我们同样要在最后将 $\boldsymbol{w}$ 恢复为零向量。

在稀疏向量加法中，非零元有序的版本会导致大量插入操作，容易得不偿失。在稀疏向量乘法中，非零元有序的版本会简洁很多。因此，如果稀疏向量需要经常修改，则非零元无序可能会更好，而如果稀疏向量不需要修改，则非零元有序可能会更好。

4.6.2 稀疏矩阵

与稠密矩阵相比，稀疏矩阵的存储方式更为多样化。这一方面使得我们有更多的机会提高程序的效率，另一方面也使得数据结构的选择更为困难，程序的逻辑更为复杂。在编写程序前，我们应当汇总所需要的矩阵操作，分析每一个数据结构对于这些操作的效率，再做出最为合适的选择。

本节我们使用多种不同的方式来存储这样一个示例矩阵：

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 3 & 5 \\ 4 & 0 & 7 \end{pmatrix} \tag{4.35}$$

1. COO（Coordinate List）

COO 格式也称为三元组格式。它是指存储（行标, 列标, 值）这样一个三元组的列表，如表 4.5所示。三元组可以按照某种顺序排序以便于今后处理，这样的数据结构非常直观，增加一个元素及遍历矩阵元素都非常简单，但查找元素和修改元素则非常困难。

表 4.5 COO 数据结构示例

行　标	列　标	值
1	1	1
2	2	3
2	3	5
3	1	4
3	3	7

2. DOK（Dictionary Of Keys）

DOK 格式就是将行标和列标的组合映射到元素值。如表 4.6所示。这样的数据结构也非常直观。与 COO 相比，DOK 可以在更短的时间内查找和修改元素。但遍历元素的效率很低。

表 4.6 DOK 数据结构示例

(行标, 列标)		值
(1,1)	→	1
(2,2)	→	3
(2,3)	→	5
(3,1)	→	4
(3,3)	→	7

3. LIL（List of List）

LIL 格式对每一行存储一个列表，其中包含列标和元素值。我们也可以理解成每一行都存储为一个稀疏向量。这里的稀疏向量可以采用之前提到过的方式存储，如表 4.7所示。与稀疏向量类似，这里的每一行也可以选择是否要求列标按顺序存储。

表 4.7　LIL 数据结构示例（按行）

行　标	(列标, 元素)，(列标, 元素)，···
1	(1,1)
2	(2,3), (3,5)
3	(1,4), (3,7)

相对应地，也可以是每一列存储一个列表，其中包含行标和元素值。如表4.8所示。

表 4.8　LIL 数据结构示例（按列）

列　标	(行标, 元素)，(行标, 元素)，···
1	(1,1), (3,4)
2	(2,3)
3	(2,5), (3,7)

LIL 格式的速度通常会比 DOK 格式更快，这是因为字典格式本身容易产生效率瓶颈。因此在许多软件中，LIL 格式都被选为主要数据结构之一。

4. CSR（Compressed Sparse Row）

CSR 格式是指行压缩格式。我们连续存储同一行的元素，并把每一行按顺序接在一起。表 4.9给出了 CSR 数据结构的一个示例。其中行起止位置表示的是每一行的列标和元素的起始位置。例如 $0,1,3,5$ 代表第 1 行的起止位置是 $0,1$，第 2 行的起止位置是 $1,3$，第 3 行的起止位置是 $3,5$。也就是说，第 1 个列标和元素是第 1 行的，第 2、3 个列标和元素是第 2 行的，第 4、5 个列标和元素是第 3 行的。$m \times n$ 的矩阵，行起止位置需要存储 $m+1$ 个数，但其中第 1 个数总是 0。我们存储 0 作为第 1 个数的目的是写程序更方便。

表 4.9　CSR 数据结构示例

行起止位置	0, 1, 3, 5
列　标	1, 2, 3, 1, 3
元　素	1, 3, 5, 4, 7

5. CSC（compressed sparse column）

CSC 格式是指列压缩格式，同 CSR 格式类似。如表 4.10所示。

对于稀疏矩阵，我们这里已经介绍了许多常见的数据结构。其中 LIL、CSR 和 CSC 格式在实际情况中使用得较多。而 CSR 和 CSC 相对于 LIL 而言，存储更为紧密。因此，在矩阵需要修改的时候选择 LIL 格式的优势更大，在矩阵不需要修改的时候选择 CSR 和 CSC 的优势更大。实际场景往往非常复杂，最终数据结构的选择可能还需要综合考虑许多其他方面。

表 4.10　CSC 数据结构示例

列起止位置	0, 2, 3, 5
行　标	1, 3, 2, 2, 3
元　素	1, 4, 3, 5, 7

在 LU 分解的计算过程中，由于涉及修改操作，所以我们常常使用按列的 LIL 格式存储 $\boldsymbol{L}$，使用按行的 LIL 格式存储 $\boldsymbol{U}$。

4.7　稀疏 LU 分解

在本节中，我们考虑在 LU 分解中尽量少引入新的非零元素，也就是说在 LU 分解中尽量维持稀疏度。为了描述方便，当我们主要关注稀疏度而不是数值稳定性的时候，常常简单地将矩阵中的非零元素标记为星号，去研究如何在 LU 分解时尽量不增加星号数量。由于在实际计算过程中会出现抵消，所以星号并不一定始终是非零元素。但这种情况毕竟属于少数，对稀疏度的影响非常小，可以直接忽略。

4.7.1　Markowitz 算法

Markowitz 算法是由美国经济学家 Harry Max Markowitz 教授提出的[Mar57]。Markowitz 教授曾在 1989 年获得冯诺依曼理论奖，在 1990 年获得诺贝尔经济学奖。

Markowitz 算法基于这样一个观察：假设在活跃子矩阵中，主元所在行上的非零元共有 r 个，主元所在列上的非零元共有 c 个，则在最坏情况下会增加 $(r-1)(c-1)$ 个非零元素。我们来看一个具体的例子。假设我们有一个 7 阶方阵，第 2 行第 2 列是活跃子矩阵的左上角，如果我们选择第 2 行第 2 列为主元，则引入非零元素的情况如下画线标注所示。

$$\begin{pmatrix} * & * & * & & & * & \\ & * & * & & & * & * \\ & & * & & * & * & \\ * & & & * & & & \\ & & & * & & & * \\ * & * & & * & & & \\ & & * & & & & \end{pmatrix} \rightarrow \begin{pmatrix} * & * & * & & & * & \\ & * & * & & & * & * \\ & & * & & * & * & \\ * & \underline{*} & * & & \underline{*} & \underline{*} & \\ & & & * & & * & \\ * & \underline{*} & & * & \underline{*} & \underline{*} & \\ & & * & & & * & \end{pmatrix} \tag{4.36}$$

注意这里下画线所标注的元素也有一些原本就是非零元素。精确计算引入非零元素的数量是得不偿失的，所以 Markowitz 算法使用了 $(r-1)(c-1)$ 这个上界来粗略估计。在这个例子中，$r=3$，$c=4$（注意第 1 行和第 1 列不属于活跃子矩阵），因此方框个数是 $(r-1)(c-1)=6$ 个，真实引入的非零元素不超过 6 个。

Markowitz 算法的核心思想就是选择一个主元，使得评价函数 $m(i,j)=(r-1)(c-1)$ 达到最小。一个特殊情形是 $r=1$ 或者 $c=1$ 时 $m(i,j)=0$，即不会产生任何新的非零元素。这里 $r=1$ 意味着主元是活跃子矩阵中其所在行上的唯一非零元素，$c=1$ 意味着主元是活跃子矩阵中其所在列上的唯一非零元素。

在实际使用 Markowitz 方法时，我们还需要考虑数值稳定性和计算效率，因此我们还需要注意两点。

- 主元不能太小。我们可以通过一个参数来控制主元的下界，避免出现精度问题。
- 求主元的计算量不能太大。对活跃子矩阵中的每个非零元素计算 $(r-1)(c-1)$ 仍然是非常费时的。我们可以限制主元仅能在元素较少的行或列中选取，这样就可以直接跳过元素较多的行或列。

4.7.2 最小度算法

我们之前曾针对半正定矩阵介绍了 Cholesky 分解。本节要介绍的最小度算法也是针对半正定矩阵的特殊性而设计的，主要用于选择主元。它实际上就是 Markowitz 算法的对称版本，但由于对称这一特殊性，可以与图论结合起来理解。对称版本的 Markowitz 算法最早是由 [TW67] 提出的，而后被 [Ros72] 以图论的角度描述为最小度算法。

对称矩阵都可以与无向图一一对应。例如矩阵

$$\begin{pmatrix} * & & * & * & & & \\ & * & & & & * & \\ * & & * & & * & * & \\ * & & & * & & & \\ & & * & & * & & \\ & * & * & & & * & \end{pmatrix} \tag{4.37}$$

可以与图 4.1对应。我们可以认为矩阵对角线上都是非零元素，这些位置对应于图的顶点。在非对角线上，矩阵第 i 行第 j 列是非零元素，就意味着图中第 i 个顶点和第 j 个顶点之间

有边相连，是零元素就意味着图中第 i 个顶点和第 j 个顶点之间无边相连。因此每一条边都代表了一个非零元。

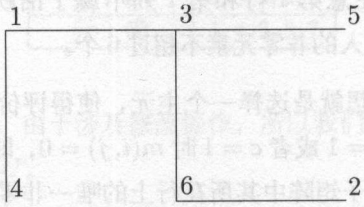


图 4.1 对称矩阵对应的无向图示例

建立了一一对应关系之后，我们就可以用图来描述活跃子矩阵，用顶点和边的改变来描述消去过程。注意在选择主元的时候，为了保证矩阵的对称性，我们只能选择对角线上的元素，而对角线上的元素对应图的顶点。每次选中一个主元后，我们从图中删除相应的顶点和与其相关的所有边，同时对其所有相邻的点两两连边。高斯消去法就相当于反复进行这样的操作，直到删完所有顶点。每次连新边就相当于引入了新的非零元，因此为了减少引入非零元的数量，我们可以每次挑选图中度数最小的顶点。这样的图操作与高斯消去法是对应的，因此我们称之为消去图（Elimination Graph）。虽然算法描述非常简单，但不幸的是，它并不适合直接使用。这主要是由于图中的边不断增加，对存储空间和计算时间都带来了巨大的压力。

下面我们引入商图（Quotient Graph）来代替消去图。在消去图中，我们显式地表示边。为了降低空间复杂度，商图改为隐式表示边。在删除顶点的时候，我们不真的删除顶点，只是把其标记为已删顶点。如果两个已删除的顶点相邻，那么我们可以把它们合并成一个新的已删除的顶点，它们的邻居也合并成为新点的邻居。在计算给定顶点的度数时，我们要计算它的邻居的数量，如果某个邻居是已删除的顶点，则我们也要再计算已删除的顶点的邻居，注意这里要避免重复计数。举个例子，考虑矩阵

$$\begin{pmatrix} * & * & * & * & * & * \\ * & * & & & & \\ * & & * & & & \\ * & & & * & & \\ * & & & & * & \\ * & & & & & * \end{pmatrix} \tag{4.38}$$

所对应的无向图如图 4.2左侧所示。假设我们以第 1 行第 1 列为主元进行消去，也就是说删除图 4.2中编号为 1 的顶点，则对应的消去图是 5 个顶点的完全图，如图 4.2右上方所示。而对应的商图则几乎没有任何变化，只是把编号为 1 的顶点进行标记，如图 4.2右下方所示。这样的技巧并不影响算法结果，但可以大大减少计算量。

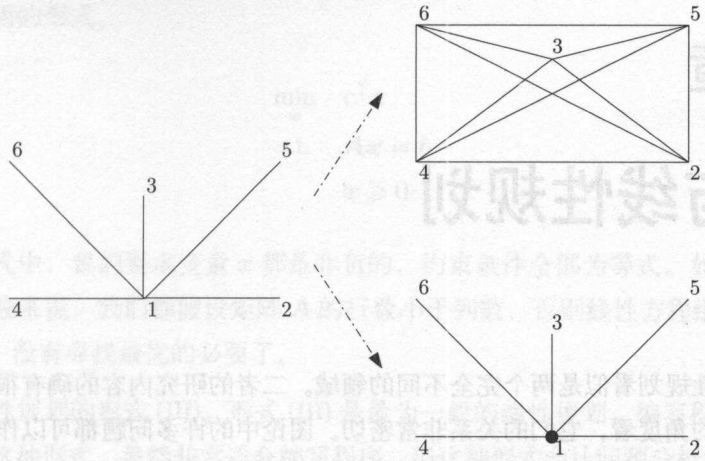


图 4.2 消去图与商图对比示例

如果在图 4.2中有两个邻居顶点，它们的其他邻居是完全一样的，那么这两个顶点本质上是相同的。因为无论删除图中其他任何一个顶点，这两个顶点还是等价的。如果删除这两个点中的其中一个，那么下一步就一定可以删除另外一个顶点。因此最简单、快速的办法就是合并这两个顶点。同理我们也可以合并多个顶点。这样在消去的过程中，我们就可以一步同时消去这些顶点了。我们还需要给每个顶点记录一个权重，初始时每个顶点的权重都是 1。当合并顶点之后，新的顶点的权重就是被合并点的权重之和，这样可以方便我们正确计算顶点的度数。

在实际应用中，原始的最小度算法并不会被经常使用，而 Multiple Minimum Degree（MMD）算法和 Approximate Minimum Degree（AMD）算法被经常使用。这两种算法都改进了更新度数的方法，提高了运行效率。在 MMD 算法中对于一组有相同度数但并不是邻居的顶点，仍然合并在一起一步消除。这样要进行的消去步数减少，整体运行时间降低。在 AMD 算法中不再精确计算度数，而是使用估算的上界：一个顶点的度数比它所有邻居（普通顶点和已删顶点）的权重之和小。在测试中发现，AMD 算法一般会得到更少的非零元，以及花费更少的时间得到一个较优解。

第 5 章

图论与线性规划

图论与线性规划看似是两个完全不同的领域。二者的研究内容的确有很大的差别，但如果仅从算法的角度看，它们的关系非常密切。图论中的许多问题都可以作为线性规划问题或更为一般的最优化问题。尽管图论中的一些经典问题已经有非常成熟的特定算法，但线性规划作为一个更具有一般性的工具，可以用来理解图论中的已有算法和发现图论中的新算法。

在本章中我们首先介绍线性规划的基本概念和算法，然后通过关联矩阵、全单模矩阵将图论与线性规划联系起来，最后介绍若干图论中的经典问题，并结合线性规划进行理解。

5.1 线性规划基础

线性规划是指在一些线性约束的条件下，求解线性目标函数的最优值。线性约束、线性目标函数的具体写法可能包含多种形式，在不同的书和论文中可能会根据具体情况选择便于分析的某一种形式。之所以可以选择某一种形式进行分析，是因为这些不同形式的线性规划是等价的。我们这里列举出其中最为常见的几种形式，并说明这些形式的等价性。

首先是线性规划的形式 (I)。

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \geqslant \boldsymbol{b} \end{aligned} \tag{5.1}$$

在这个形式中，我们对变量 $\boldsymbol{x}$ 本身没有限制，约束条件全部为不等式。如果不考虑退化的情形，则一般来说，我们都假设矩阵 $\boldsymbol{A}$ 的行数大于列数，否则约束数量太少，无法限制变量 $\boldsymbol{x}$ 的范围，常常会导致最优值无穷。

其次是线性规划的形式 (II)，在许多书中也叫作线性规划的标准形式，这也是介绍单纯形方法时最常用的形式。

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \geqslant 0 \end{aligned} \tag{5.2}$$

在这个形式中，我们要求变量 $\boldsymbol{x}$ 都是非负的，约束条件全部为等式。如果不考虑退化的情形，则一般来说，我们都假设矩阵 $\boldsymbol{A}$ 的行数小于列数，否则线性方程组 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ 就足以限制变量 $\boldsymbol{x}$，没有寻找最优的必要了。

最后是线性规划的形式 (III)。形式 (III) 是最为一般的线性规划，编写程序时实际存储所采用的也是这种形式，虽然非常适合编写程序，但这种形式会让问题分析复杂化。

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \\ \text{s.t.} \quad & \underline{\boldsymbol{b}} \leqslant \boldsymbol{A}\boldsymbol{x} \leqslant \overline{\boldsymbol{b}} \\ & \underline{\boldsymbol{x}} \leqslant \boldsymbol{x} \leqslant \overline{\boldsymbol{x}} \end{aligned} \tag{5.3}$$

其中 $\underline{\boldsymbol{b}}$、$\overline{\boldsymbol{b}}$、$\underline{\boldsymbol{x}}$、$\overline{\boldsymbol{x}}$ 都是输入常数并且可以是无穷。

容易看出，形式 (III) 涵盖了线性规划的所有表述方式。例如，线性规划的形式 (I) 和形式 (II) 都可以直接看作形式 (III) 的特例。

- 若 $\overline{\boldsymbol{b}} = +\infty, \underline{\boldsymbol{x}} = -\infty, \overline{\boldsymbol{x}} = +\infty$，则形式 (III) 变为形式 (I)。
- 若 $\underline{\boldsymbol{b}} = \overline{\boldsymbol{b}}, \underline{\boldsymbol{x}} = 0, \overline{\boldsymbol{x}} = +\infty$，则形式 (III) 变为形式 (II)。

另外，如果目标函数是 $\max \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}$，则我们可以定义 $\boldsymbol{c}' = -\boldsymbol{c}$，于是目标函数等价于 $\min \boldsymbol{c}'^{\mathrm{T}}\boldsymbol{x}$，与形式 (III) 一致。

形式 (I) 和形式 (II) 看上去比形式 (III) 要简单很多，但它们同样是等价的。我们首先给出形式 (III) 转换为形式 (I) 的步骤。形式 (III) 可以等价地改写为

$$
\begin{aligned}
\min_{\boldsymbol{x}} \quad & \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \\
\text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \geqslant \underline{\boldsymbol{b}} \\
& -\boldsymbol{A}\boldsymbol{x} \geqslant -\overline{\boldsymbol{b}} \\
& \boldsymbol{x} \geqslant \underline{\boldsymbol{x}} \\
& \boldsymbol{x} \geqslant -\overline{\boldsymbol{x}}
\end{aligned}
\tag{5.4}
$$

所以，如果我们记

$$
\boldsymbol{A}' = \begin{pmatrix} \boldsymbol{A} \\ -\boldsymbol{A} \\ \boldsymbol{I} \\ -\boldsymbol{I} \end{pmatrix}, \quad \boldsymbol{b}' = \begin{pmatrix} \underline{\boldsymbol{b}} \\ -\overline{\boldsymbol{b}} \\ \underline{\boldsymbol{x}} \\ -\overline{\boldsymbol{x}} \end{pmatrix}
\tag{5.5}
$$

那么形式 (III) 变为

$$
\begin{aligned}
\min_{\boldsymbol{x}} \quad & \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \\
\text{s.t.} \quad & \boldsymbol{A}'\boldsymbol{x} \geqslant \boldsymbol{b}'
\end{aligned}
\tag{5.6}
$$

我们再给出形式 (I) 转换为形式 (II) 的步骤。很显然，任意实数 x 都可以写为两个非负数的差，即 $x^+ - x^-$，其中 $x^+ \geqslant 0$，$x^- \geqslant 0$。因此，形式 (I) 可以改写为

$$
\begin{aligned}
\min_{\boldsymbol{x}^+, \boldsymbol{x}^-} \quad & \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}^+ - \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}^- \\
\text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x}^+ - \boldsymbol{A}\boldsymbol{x}^- \geqslant \boldsymbol{b} \\
& \boldsymbol{x}^+ \geqslant 0 \\
& \boldsymbol{x}^- \geqslant 0
\end{aligned}
\tag{5.7}
$$

我们可以再引入一个新的变量 $\boldsymbol{x}^r$，满足 $\boldsymbol{x}^r = \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b} = \boldsymbol{A}\boldsymbol{x}^+ - \boldsymbol{A}\boldsymbol{x}^- - \boldsymbol{b} \geqslant 0$。这样形式 (I) 进一步改写为

$$
\begin{aligned}
\min_{\boldsymbol{x}^+, \boldsymbol{x}^-, \boldsymbol{x}^r} \quad & \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}^+ - \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}^- \\
\text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x}^+ - \boldsymbol{A}\boldsymbol{x}^- - \boldsymbol{x}^r = \boldsymbol{b} \\
& \boldsymbol{x}^+ \geqslant 0 \\
& \boldsymbol{x}^- \geqslant 0 \\
& \boldsymbol{x}^r \geqslant 0
\end{aligned}
\tag{5.8}
$$

所以，如果我们记

$$\boldsymbol{A}' = (\boldsymbol{A}, -\boldsymbol{A}, -\boldsymbol{I}), \boldsymbol{c}' = \begin{pmatrix} \boldsymbol{c} \\ -\boldsymbol{c} \\ 0 \end{pmatrix}, \boldsymbol{x}' = \begin{pmatrix} \boldsymbol{x}^+ \\ \boldsymbol{x}^- \\ \boldsymbol{x}^r \end{pmatrix} \tag{5.9}$$

那么形式 (I) 就变为

$$\begin{aligned} \min_{\boldsymbol{x}'} \quad & \boldsymbol{c}'^{\mathrm{T}}\boldsymbol{x}', \\ \text{s.t.} \quad & \boldsymbol{A}'\boldsymbol{x}' = \boldsymbol{b} \\ & \boldsymbol{x}' \geqslant 0 \end{aligned} \tag{5.10}$$

因此这三种线性规划的形式是互相等价的，对于任意一个线性规划问题，也可以根据需要改写为其中一种形式。为了方便，我们在后面常常会选一种形式进行推导，对于其余形式，感兴趣的读者可以根据等价性做类似的推导。

5.1.1 Fourier Motzkin 消去法

Fourier Motzkin 消去法实际上是用于求解线性不等式组的一种方法，但也可以用来求解线性规划，因为二者其实非常相似。例如在线性规划的形式 (I) 中，如果不考虑目标函数，则实际上就是求解线性不等式组。从简单性出发，我们可以先考虑求解 $\boldsymbol{x}$ 满足 $\boldsymbol{A}\boldsymbol{x} \geqslant \boldsymbol{b}$ 的这样一个线性不等式组。

我们先考虑消去变量 x_1，可以根据 x_1 在每一个不等式 $\boldsymbol{a}_i^{\mathrm{T}}\boldsymbol{x} \geqslant b_i$ 中的系数将不等式分为 3 类。

- 若 x_1 前系数为正，则可以改写为 $x_1 \geqslant \dfrac{b_i - \sum_{j=2}^{n} a_{i,j}x_j}{a_{i,1}}$。
- 若 x_1 前系数为负，则可以改写为 $x_1 \leqslant \dfrac{b_i - \sum_{j=2}^{n} a_{i,j}x_j}{a_{i,1}}$。
- 若 x_1 前系数为零，则不等式和 x_1 无关，无须改写。

我们记这 3 类不等式的行号分别属于集合 $\boldsymbol{I}^+$、$\boldsymbol{I}^-$、$\boldsymbol{I}^0$，则对于任意 $i^+ \in \boldsymbol{I}^+, i^- \in \boldsymbol{I}^-$，根据改写的形式，我们直接得到

$$\frac{b_{i^-}-\sum_{j=2}^{n}a_{i^-,j}x_j}{a_{i^-,1}}\geqslant x_1\geqslant\frac{b_{i^+}-\sum_{j=2}^{n}a_{i^+,j}x_j}{a_{i^+,1}}$$

这样我们就可以消去 x_1，直接写为

$$\frac{b_{i^-}-\sum_{j=2}^{n}a_{i^-,j}x_j}{a_{i^-,1}}\geqslant\frac{b_{i^+}-\sum_{j=2}^{n}a_{i^+,j}x_j}{a_{i^+,1}}$$

我们可以使用这种方法得到 $|\boldsymbol{I}^+|\times|\boldsymbol{I}^-|$ 个新的线性不等式。只要这些不等式都满足，则 x_1 是一定存在的。因此我们将 $\boldsymbol{I}^+$ 和 $\boldsymbol{I}^-$ 中的不等式替换为这 $|\boldsymbol{I}^+|\times|\boldsymbol{I}^-|$ 个新的不等式，并保留 $\boldsymbol{I}^0$ 中的不等式。若原本共有 m 个不等式，则在消去 x_1 后，我们会得到至多 $m^2/4$ 个不等式。

在这一过程中可能会出现两种特殊情况。

- $\boldsymbol{I}^+$ 或者 $\boldsymbol{I}^-$ 为空。这时我们可以丢弃所有包含 x_1 的不等式。若 $\boldsymbol{I}^+$ 为空，则说明 x_1 可以非常小，使得所有含 x_1 的不等式成立。
- 出现某个不等式不包含任何变量。这时如果不等式成立则可以删去，否则说明问题无解。

类似地，我们可以依次消去 $x_2,x_3,\cdots,x_{n-1}$，最终得到仅包含 x_n 的线性不等式组。不等式的数量至多会有 m^{2^n} 个。这时我们容易得到 x_n 的一个取值，并根据消去过程反推出前面的各个变量。

即使对于包含目标函数的线性规划，我们也可以应用 Fourier Motzkin 消去法。我们考虑线性规划的形式 (I)，如果引入一个新的变量，则可以将其改写为

$$\begin{aligned}\min\quad & x_{n+1}\\ \text{s.t.}\quad & \boldsymbol{A}\boldsymbol{x}\geqslant\boldsymbol{b}\\ & x_{n+1}-\boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}\geqslant 0\end{aligned}\tag{5.11}$$

我们可以对不等式组

$$\begin{cases}\boldsymbol{A}\boldsymbol{x}\geqslant\boldsymbol{b}\\ x_{n+1}-\boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}\geqslant 0\end{cases}\tag{5.12}$$

应用 Fourier Motzkin 消去法，依次消去 $x_1, x_2, \cdots, x_n$，最终得到仅包含 x_{n+1} 的不等式组。此时容易求得 x_{n+1} 的最小值，即目标函数的最优值。虽然 Fourier Motzkin 消去法可以用于求解任意的线性规划问题，但即使对于规模并不大的问题，其计算量也可能是巨大的。在实际问题中虽然很少使用，但这种方法给了我们一种很有意义的思路：可以保留一部分变量，消去另一部分。这和高斯消元法很相似，同时有助于我们在后面理解线性规划的基变量。

后面将要介绍的单纯形方法，其思想与 Fourier Motzkin 消去法也有一些相似之处。

5.1.2 基

我们考虑线性规划的形式 (II)，即标准形式

$$\begin{aligned} \min \quad & \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \geqslant 0 \end{aligned} \tag{5.13}$$

其中 $\boldsymbol{A}$ 是 m 行 n 列的矩阵。不考虑退化情况，我们一般假设 $m < n$。

基的引入将线性规划同线性方程组紧密联系在一起。基实质上就是将变量分为两类，一类是取到上界或下界，另一类是没有取到上界或下界。具体反映在线性规划的形式 (II) 中，也就是以变量是否取到 0 为标准。如果我们选出 $n - m$ 个变量都强制设为 0，那么 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ 这 m 个等式实质上就只剩下 m 个变量了。也就是说很可能 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ 有唯一解。我们甚至可以用之前介绍过的高斯消去法来求解这 m 个变量。

我们定义集合 $\mathcal{B} \subseteq \{1, 2, \cdots, n\}$ 且 $|\mathcal{B}| = m$ 是没有取到 0 的变量的下标集合，$\bar{\mathcal{B}} = \{1, 2, \cdots, n\} - \mathcal{B}$ 是取到 0 的变量的下标集合。我们称 $\mathcal{B}$ 是基，$\bar{\mathcal{B}}$ 是非基。相应地，我们还可以做如下定义。

- $\boldsymbol{x}_{\mathcal{B}}$ 表示 $\mathcal{B}$ 中下标所对应的行拼成的列向量，$\boldsymbol{x}_{\bar{\mathcal{B}}}$ 表示 $\bar{\mathcal{B}}$ 中下标所对应的行拼成的列向量，我们称 $\boldsymbol{x}_{\mathcal{B}}$ 为基变量，称 $\boldsymbol{x}_{\bar{\mathcal{B}}}$ 为非基变量。
- $\boldsymbol{A}_{\mathcal{B}}$ 表示 $\mathcal{B}$ 中下标所对应的列拼成的子矩阵，$\boldsymbol{A}_{\bar{\mathcal{B}}}$ 表示 $\bar{\mathcal{B}}$ 中下标所对应的列拼成的子矩阵。

那么我们就有 $\boldsymbol{x}_{\bar{\mathcal{B}}} = 0$，约束 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ 就变为 $\boldsymbol{A}_{\mathcal{B}}\boldsymbol{x}_{\mathcal{B}} + \boldsymbol{A}_{\bar{\mathcal{B}}}\boldsymbol{x}_{\bar{\mathcal{B}}} = \boldsymbol{A}_{\mathcal{B}}\boldsymbol{x}_{\mathcal{B}} = \boldsymbol{b}$。因此如果 $\boldsymbol{A}_{\mathcal{B}}$ 可逆，则我们就有 $\boldsymbol{x}_{\mathcal{B}} = \boldsymbol{A}_{\mathcal{B}}^{-1}\boldsymbol{b}$。请注意这时 $\boldsymbol{x}_{\mathcal{B}}$ 并不一定满足 $\boldsymbol{x}_{\mathcal{B}} \geqslant 0$ 的约束条件，如果满足 $\boldsymbol{x}_{\mathcal{B}} \geqslant 0$，那么我们称这样的一个可行解 $\boldsymbol{x}$ 为基本可行解。

对于基，有这样的一个重要的定理。

定理 5.1 对于任意标准形式的线性规划，以下三种情况必有一种成立。

- 无可行解。
- 最优值是无穷。
- 有一个最优解是基本可行解。

我们略去其证明。单从结论上看，无可行解和最优值是无穷都是非常特殊的情况，而对于一般情况而言，求解线性规划只需从基本可行解中寻求最优。

我们在前面定义的基是针对线性规划的形式 (II)。对于其他形式的线性规划，我们需要先将线性不等式约束（不包括变量上下界约束）都变为线性等式约束。例如，对于线性规划的形式 (I)，我们可以引入新的变量 $\boldsymbol{r}$，并令 $\boldsymbol{r} = \boldsymbol{A}\boldsymbol{x}$。这时线性规划的形式 (I) 就等价于

$$\begin{aligned} \min_{\boldsymbol{x},\boldsymbol{r}} \quad & \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} - \boldsymbol{r} = 0 \\ & \boldsymbol{r} \geqslant \boldsymbol{b} \end{aligned} \tag{5.14}$$

根据前面的定义，非基变量要取到上界或下界，因此这里只有 $\boldsymbol{r}$ 中的变量可以成为非基变量，而 $\boldsymbol{x}$ 中的变量则一定为基变量。如果 $\boldsymbol{A}$ 是 m 行 n 列的矩阵，则由于有 m 个等式，基变量的数目也应当是 m 个，相应的非基变量的数目就应当是 n 个。而对于线性规划的形式 (III)，我们同样引入新的变量 $\boldsymbol{r}$，并令 $\boldsymbol{r} = \boldsymbol{A}\boldsymbol{x}$。这时线性规划的形式 (III) 就等价于

$$\begin{aligned} \min_{\boldsymbol{x},\boldsymbol{r}} \quad & \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} - \boldsymbol{r} = 0 \\ & \underline{\boldsymbol{b}} \leqslant \boldsymbol{r} \leqslant \overline{\boldsymbol{b}} \\ & \underline{\boldsymbol{x}} \leqslant \boldsymbol{x} \leqslant \overline{\boldsymbol{x}} \end{aligned} \tag{5.15}$$

这里，只要上界或下界不是无穷，则对应的变量就都有可能成为非基变量。特别是，当上界和下界都不是无穷时，非基变量既有可能取上界，也有可能取下界。如果 $\boldsymbol{A}$ 是 m 行 n 列的矩阵，则由于有 m 个等式，基变量的数目也应当是 m 个，相应的非基变量的数目就应当是 n 个。

5.1.3 单纯形方法

单纯形方法是 1947 年由美国数学家 George Bernard Dantzig 提出的。Dantzig 教授曾在 1989 年获得冯诺依曼理论奖。关于 Dantzig 教授，有一段很有意思的故事。Dantzig 师从著名的统计学家 Jerzy Neyman，有一次 Dantzig 上课迟到了，把黑板上的两道题目误以为是作业而记了下来，后来他花了好几天才做好，当他交给 Neyman 教授之后才知道这原来是著名的未解决的问题。

单纯形方法的思想非常简单：不断调整基本可行解，最终得到最优解。我们仍然以线性规划的形式 (II) 为例，假设已经有了一个初始基 $\mathcal{B}$ 及相应的非基 $\bar{\mathcal{B}}$，则 $\boldsymbol{x}_{\mathcal{B}} = \boldsymbol{A}_{\mathcal{B}}^{-1}(\boldsymbol{b} - \boldsymbol{A}_{\bar{\mathcal{B}}}\boldsymbol{x}_{\bar{\mathcal{B}}})$，于是目标函数满足

$$\begin{aligned} \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} &= \boldsymbol{c}_{\mathcal{B}}^{\mathrm{T}}\boldsymbol{x}_{\mathcal{B}} + \boldsymbol{c}_{\bar{\mathcal{B}}}^{\mathrm{T}}\boldsymbol{x}_{\bar{\mathcal{B}}} \\ &= \boldsymbol{c}_{\mathcal{B}}^{\mathrm{T}}\boldsymbol{A}_{\mathcal{B}}^{-1}(\boldsymbol{b} - \boldsymbol{A}_{\bar{\mathcal{B}}}\boldsymbol{x}_{\bar{\mathcal{B}}}) + \boldsymbol{c}_{\bar{\mathcal{B}}}^{\mathrm{T}}\boldsymbol{x}_{\bar{\mathcal{B}}} \\ &= \boldsymbol{c}_{\mathcal{B}}^{\mathrm{T}}\boldsymbol{A}_{\mathcal{B}}^{-1}\boldsymbol{b} + (\boldsymbol{c}_{\bar{\mathcal{B}}} - \boldsymbol{A}_{\bar{\mathcal{B}}}^{\mathrm{T}}\boldsymbol{A}_{\mathcal{B}}^{-\mathrm{T}}\boldsymbol{c}_{\mathcal{B}})^{\mathrm{T}}\boldsymbol{x}_{\bar{\mathcal{B}}} \end{aligned} \tag{5.16}$$

注意到第 1 部分 $\boldsymbol{c}_{\mathcal{B}}^{\mathrm{T}}\boldsymbol{A}_{\mathcal{B}}^{-1}\boldsymbol{b}$ 不包含变量，第 2 部分变量 $\boldsymbol{x}_{\bar{\mathcal{B}}}$ 的系数是 $\boldsymbol{c}_{\bar{\mathcal{B}}} - \boldsymbol{A}_{\bar{\mathcal{B}}}^{\mathrm{T}}\boldsymbol{A}_{\mathcal{B}}^{-\mathrm{T}}\boldsymbol{c}_{\mathcal{B}}$。由于在约束中包含 $\boldsymbol{x}_{\bar{\mathcal{B}}} \geqslant 0$，如果系数满足 $\boldsymbol{c}_{\bar{\mathcal{B}}} - \boldsymbol{A}_{\bar{\mathcal{B}}}^{\mathrm{T}}\boldsymbol{A}_{\mathcal{B}}^{-\mathrm{T}}\boldsymbol{c}_{\mathcal{B}} \geqslant 0$，那么显然 $\boldsymbol{x}_{\bar{\mathcal{B}}}$ 取 0 时目标函数最小；相反，如果 $\boldsymbol{c}_{\bar{\mathcal{B}}} - \boldsymbol{A}_{\bar{\mathcal{B}}}^{\mathrm{T}}\boldsymbol{A}_{\mathcal{B}}^{-\mathrm{T}}\boldsymbol{c}_{\mathcal{B}}$ 中有某个分量为负，则说明对应的 $\boldsymbol{x}_{\bar{\mathcal{B}}}$ 中的分量如果比 0 稍大一点，就会导致目标函数有改进。因此我们有如下定理。

定理 5.2（基的最优性） 若基的选取满足 $\boldsymbol{c}_{\bar{\mathcal{B}}} - \boldsymbol{A}_{\bar{\mathcal{B}}}^{\mathrm{T}}\boldsymbol{A}_{\mathcal{B}}^{-\mathrm{T}}\boldsymbol{c}_{\mathcal{B}} \geqslant 0$，则基本可行解是最优解，反之则不是最优解。

如果基本可行解不是最优解，则根据上述分析，我们可以找到一个 $j \in \bar{\mathcal{B}}$，当 x_j 增大时，目标函数会有改进。这时我们应当把 j 从 $\bar{\mathcal{B}}$ 中取出并放入 $\mathcal{B}$ 中，相应地，也必须从 $\mathcal{B}$ 中取出一个下标放入 $\bar{\mathcal{B}}$ 中来维持基的定义。假设 x_j 从 0 变为 α，则其余非基变量保持为 0。约束条件 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ 等价于 $\boldsymbol{x}_{\mathcal{B}} + \boldsymbol{A}_{\mathcal{B}}^{-1}\boldsymbol{A}_{\bar{\mathcal{B}}}\boldsymbol{x}_{\bar{\mathcal{B}}} = \boldsymbol{A}_{\mathcal{B}}^{-1}\boldsymbol{b}$，因此 x_j 从 0 变为 α 后，应当有 $\boldsymbol{x}_{\mathcal{B}} + \alpha\boldsymbol{A}_{\mathcal{B}}^{-1}\boldsymbol{A}_j = \boldsymbol{A}_{\mathcal{B}}^{-1}\boldsymbol{b}$。为了保持可行性，即 $\boldsymbol{x}_{\mathcal{B}} \geqslant 0$，应当有 $\alpha\boldsymbol{A}_{\mathcal{B}}^{-1}\boldsymbol{A}_j \leqslant \boldsymbol{A}_{\mathcal{B}}^{-1}\boldsymbol{b}$，因此 $\alpha \leqslant \min\{\dfrac{(\boldsymbol{A}_{\mathcal{B}}^{-1}\boldsymbol{b})_i}{(\boldsymbol{A}_{\mathcal{B}}^{-1}\boldsymbol{A}_j)_i} \mid (\boldsymbol{A}_{\mathcal{B}}^{-1}\boldsymbol{A}_j)_i > 0\}$。当 α 取到这个上界的时候，会有至少一个 $i \in \mathcal{B}$ 满足 $x_i = 0$，于是可以将其从 $\mathcal{B}$ 中取出。如果 α 无上界，则说明 x_j 可以无限增大，目标函数可以无限改进，也就是说最优值为无穷。

算法 36 线性规划的单纯形方法

输入： 初始基 $\mathcal{B}$，对应的非基 $\bar{\mathcal{B}}$

1: **while** $\boldsymbol{c}_{\bar{\mathcal{B}}} - \boldsymbol{A}_{\bar{\mathcal{B}}}^{\mathrm{T}} \boldsymbol{A}_{\mathcal{B}}^{-\mathrm{T}} \boldsymbol{c}_{\mathcal{B}} \geqslant 0$ 不成立 **do**

2: 选出某个 $j \in \bar{\mathcal{B}}$ 满足 $c_j - \boldsymbol{A}_j{}^{\mathrm{T}} \boldsymbol{A}_{\mathcal{B}}^{-T} \boldsymbol{c}_{\mathcal{B}} < 0$。

3: 选出某个 $i = \arg\min_i \{ \dfrac{(\boldsymbol{A}_{\mathcal{B}}^{-1}\boldsymbol{b})_i}{(\boldsymbol{A}_{\mathcal{B}}^{-1}\boldsymbol{A}_j)_i} \mid (\boldsymbol{A}_{\mathcal{B}}^{-1}\boldsymbol{A}_j)_i > 0 \}$。若 i 不存在，则最优值为无穷。

4: 将 i 从 $\mathcal{B}$ 中取出放入 $\bar{\mathcal{B}}$，将 j 从 $\bar{\mathcal{B}}$ 中取出放入 $\mathcal{B}$。

5: **end while**

在线性规划的单纯形法中有一定的自由度，具体选择 i 和选择 j 的方法可以有很多，一般都是依靠某种贪婪策略。可惜的是，目前仍然没有任何一个单纯形方法的变种可以被证明在多项式时间内达到最优解。而线性规划的另外一个经典算法——内点法已经被证明是多项式算法。尽管如此，单纯形方法仍然占据着非常重要的地位，这主要有两方面的原因。

- 从实际表现上看，单纯形方法在某些问题上比内点法的计算速度更快，尤其是对于一些与图论相关的线性规划。而单纯形方法中的可行基在图论问题中往往具有实际意义。
- 如果两个线性规划问题的结构完全相同，则可以将其中一个线性规划的最优基作为另外一个线性规划的初始基，这时常常可以大幅度加速求解。这对于整数规划中的分支定界法至关重要。

以上我们都是假设已经有初始基，我们还需要一个算法去寻找初始可行基。事实上，我们可以构造一个辅助问题，其初始可行基是显然的，并且其最优基可以作为原问题的初始可行基。这样的话，我们就可以先求解辅助问题，然后用于求解原问题，这也就是所谓的两阶段法。我们在线性规划的形式 (II) 中引入新的变量 $\boldsymbol{r}$，构造辅助问题。为了描述方便，我们不妨假设在线性规划的形式 (II) 中有 $\boldsymbol{b} \geqslant 0$，可以通过在某些约束两边同时乘以 -1 来达到这一目的。辅助问题的构造如下。

$$\begin{aligned} \min_{\boldsymbol{x},\boldsymbol{r}} \quad & \boldsymbol{b} - \boldsymbol{r} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} - \boldsymbol{r} = 0 \\ & \boldsymbol{r} \leqslant \boldsymbol{b} \\ & \boldsymbol{x} \geqslant 0 \end{aligned} \tag{5.17}$$

我们只需选择 $\boldsymbol{r}$ 为基变量及 $\boldsymbol{x}$ 为非基变量即可得到初始可行基，此时 $\boldsymbol{x} = 0$，$\boldsymbol{r} = 0$。因目标函数 $\boldsymbol{b} - \boldsymbol{r} \geqslant 0$，所以辅助问题达到最优解时，应当有 $\boldsymbol{r} = \boldsymbol{b}$，辅助问题的最优值为 0，这时也就刚好得到了原问题的一组可行基。如果辅助问题的最优值不是 0，则说明原问题无解。

5.1.4 对偶

对偶在线性规划中是非常重要的概念。每一个线性规划问题都有一个对偶问题，并且二者具有密切的联系。我们考虑线性规划的形式 (I)，并称其为原始问题 (5.18)。

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \geqslant \boldsymbol{b} \end{aligned} \tag{5.18}$$

为了便于理解，我们将矩阵展开，写成问题 (5.19)。

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \sum_{i=1}^{n} c_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^{n} a_{1i} x_i \geqslant b_1 \\ & \sum_{i=1}^{n} a_{2i} x_i \geqslant b_2 \\ & \vdots \\ & \sum_{i=1}^{n} a_{mi} x_i \geqslant b_m \end{aligned} \tag{5.19}$$

我们知道在不等式的两边同时乘以一个非负数，不等式仍然成立。因此，对于问题 (5.19) 的任何一个可行解，以及任何 $y_j \geqslant 0$，不等式组 (5.20) 成立。

$$\begin{aligned} & \sum_{i=1}^{n} a_{1i} x_i y_1 \geqslant b_1 y_1 \\ & \sum_{i=1}^{n} a_{2i} x_i y_2 \geqslant b_2 y_2 \\ & \vdots \\ & \sum_{i=1}^{n} a_{mi} x_i y_m \geqslant b_m y_m \end{aligned} \tag{5.20}$$

将其所有不等式相加，可得

$$\sum_{i=1}^{n} (\sum_{j=1}^{m} a_{ji} y_j) x_i \geqslant \sum_{j=1}^{m} b_j y_j \tag{5.21}$$

可以看到不等式 (5.21) 的左边和目标函数 $\sum\limits_{i=1}^{n} c_i x_i$ 在形式上非常相似。如果它们的对应系数满足 $\sum\limits_{j=1}^{m} a_{ji} y_j = c_i$，那么有

$$\sum_{i=1}^{n} c_i x_i = \sum_{i=1}^{n}(\sum_{j=1}^{m} a_{ji} y_j) x_i \geqslant \sum_{j=1}^{m} b_j y_j \tag{5.22}$$

因此，我们我们可以考虑这样一个线性规划问题：

$$\begin{aligned} \max_{\boldsymbol{y}} \quad & \sum_{j=1}^{m} b_j y_j \\ \text{s.t.} \quad & \sum_{j=1}^{m} a_{j1} y_j = c_1 \\ & \sum_{j=1}^{m} a_{j2} y_j = c_2 \\ & \vdots \\ & \sum_{j=1}^{m} a_{jn} y_j = c_n \\ & y_1, y_2, \cdots, y_m \geqslant 0 \end{aligned} \tag{5.23}$$

按照之前的推导，线性规划问题 (5.23) 的最优值应当不超过问题 (5.19) 的最优值。

我们可以重新将问题 (5.23) 写成矩阵形式。

$$\begin{aligned} \max_{\boldsymbol{y}} \quad & \boldsymbol{b}^{\mathrm{T}} \boldsymbol{y} \\ \text{s.t.} \quad & \boldsymbol{A}^{\mathrm{T}} \boldsymbol{y} = \boldsymbol{c} \\ & \boldsymbol{y} \geqslant 0 \end{aligned} \tag{5.24}$$

我们称式(5.18)为线性规划原始问题，称式(5.24)为线性规划对偶问题。我们在推导过程中已经证明对偶问题的最优值不超过原始问题的最优值，这也就是所谓的弱对偶定理 (Weak Duality Theorem)。事实上，我们还有强对偶定理 (Strong Duality Theorem)，即对偶问题的最优值等于原始问题的最优值。强对偶定理的意义在于，它告诉我们原始问题和对偶问题实质上是同一个问题。因此，我们常常对比原始问题和对偶问题，选择其中较为简单的一个求解。

如果给对偶问题求对偶会是什么结果？利用类似的推导，我们可以发现对偶问题的对偶问题就是原始问题。

如果 $\boldsymbol{x}^*$ 是原始问题的最优解，$\boldsymbol{y}^*$ 是对偶问题的最优解，那么根据原始问题的最优值等于对偶问题的最优值，我们有 $\boldsymbol{c}^{\mathrm{T}} \boldsymbol{x}^* = \boldsymbol{b}^{\mathrm{T}} \boldsymbol{y}^*$。于是将它们代入式(5.22)，会刚好取到等号，即 $\boldsymbol{c}^{\mathrm{T}} \boldsymbol{x}^* = \boldsymbol{y}^{*\mathrm{T}} \boldsymbol{A} \boldsymbol{x}^* = \boldsymbol{b}^{\mathrm{T}} \boldsymbol{y}^*$。这个等式意味着

$$\begin{cases} \boldsymbol{x}^{*\mathrm{T}}(\boldsymbol{A}^{\mathrm{T}}\boldsymbol{y}^{*}-\boldsymbol{c})=0 \\ \boldsymbol{y}^{*\mathrm{T}}(\boldsymbol{A}\boldsymbol{x}^{*}-\boldsymbol{b})=0 \end{cases} \tag{5.25}$$

式(5.25)被称为线性规划的互补条件。其中第 1 个式子体现了原始问题的变量和对偶问题的约束互补，第 2 个式子体现了对偶问题的变量和原始问题的约束互补。

利用上述推导过程，我们可以得到任意形式的线性规划的对偶问题及互补条件。线性规划的形式 (I)

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x}=\boldsymbol{b} \\ & \boldsymbol{x}\geqslant 0 \end{aligned} \tag{5.26}$$

的对偶问题是

$$\begin{aligned} \max_{\boldsymbol{y}} \quad & \boldsymbol{b}^{\mathrm{T}}\boldsymbol{y} \\ \text{s.t.} \quad & \boldsymbol{A}^{\mathrm{T}}\boldsymbol{y}\leqslant\boldsymbol{c} \end{aligned} \tag{5.27}$$

互补条件是

$$\begin{cases} \boldsymbol{x}^{*\mathrm{T}}(\boldsymbol{A}^{\mathrm{T}}\boldsymbol{y}^{*}-\boldsymbol{c})=0 \\ \boldsymbol{y}^{*\mathrm{T}}(\boldsymbol{A}\boldsymbol{x}^{*}-\boldsymbol{b})=0 \end{cases} \tag{5.28}$$

线性规划的形式 (III)

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \\ \text{s.t.} \quad & \underline{\boldsymbol{b}}\leqslant\boldsymbol{A}\boldsymbol{x}\leqslant\overline{\boldsymbol{b}} \\ & \underline{\boldsymbol{x}}\leqslant\boldsymbol{x}\leqslant\overline{\boldsymbol{x}} \end{aligned} \tag{5.29}$$

的对偶问题是

$$\begin{aligned} \max_{\underline{\boldsymbol{y}},\overline{\boldsymbol{y}},\underline{\boldsymbol{z}},\overline{\boldsymbol{z}}} \quad & \underline{\boldsymbol{b}}^{\mathrm{T}}\underline{\boldsymbol{y}}-\overline{\boldsymbol{b}}^{\mathrm{T}}\overline{\boldsymbol{y}}+\underline{\boldsymbol{x}}^{\mathrm{T}}\underline{\boldsymbol{z}}-\overline{\boldsymbol{x}}^{\mathrm{T}}\overline{\boldsymbol{z}} \\ \text{s.t.} \quad & \boldsymbol{A}^{\mathrm{T}}\underline{\boldsymbol{y}}-\boldsymbol{A}^{\mathrm{T}}\overline{\boldsymbol{y}}+\underline{\boldsymbol{z}}-\overline{\boldsymbol{z}}=\boldsymbol{c} \\ & \underline{\boldsymbol{y}},\overline{\boldsymbol{y}},\underline{\boldsymbol{z}},\overline{\boldsymbol{z}}\geqslant 0 \end{aligned} \tag{5.30}$$

互补条件是

$$\begin{cases} (\boldsymbol{x}^{*}-\underline{\boldsymbol{x}})^{\mathrm{T}}\underline{\boldsymbol{z}}^{*}=0 \\ (\boldsymbol{x}^{*}-\overline{\boldsymbol{x}})^{\mathrm{T}}\overline{\boldsymbol{z}}^{*}=0 \\ (\underline{\boldsymbol{y}}^{*})^{\mathrm{T}}(\boldsymbol{A}\boldsymbol{x}^{*}-\underline{\boldsymbol{b}})=0 \\ (\overline{\boldsymbol{y}}^{*})^{\mathrm{T}}(\boldsymbol{A}\boldsymbol{x}^{*}-\overline{\boldsymbol{b}})=0 \end{cases} \tag{5.31}$$

有兴趣的读者可以自行推导得到以上结论。

5.2 全单模矩阵

5.2.1 关联矩阵

定义 5.1（关联矩阵，Incidence Matrix） 关联矩阵是用来描述两个集合间关系的矩阵。集合 $\boldsymbol{X}$ 和集合 $\boldsymbol{Y}$ 的关联矩阵是 $|\boldsymbol{X}|$ 行 $|\boldsymbol{Y}|$ 列的矩阵。如果 $\boldsymbol{X}$ 的第 i 个元素和 $\boldsymbol{Y}$ 的第 j 个元素相关，则关联矩阵第 i 行第 j 列为 1，否则为 0。

我们再具体描述一下无向图的关联矩阵和有向图的关联矩阵。无论是无向图还是有向图，其关联矩阵都描述了点集合和边集合的关系。

定义 5.2（无向图的关联矩阵） 无向图 $\boldsymbol{G}(\boldsymbol{V},\boldsymbol{E})$ 的关联矩阵是 $|\boldsymbol{V}|$ 行 $|\boldsymbol{E}|$ 列的矩阵，第 i 行第 j 列的元素 $a_{i,j}$ 满足

$$a_{i,j}=\begin{cases}1, & \text{如果第 } j \text{ 个边与第 } i \text{ 个顶点相关}\\ 0, & \text{如果第 } j \text{ 个边与第 } i \text{ 个顶点无关}\end{cases} \tag{5.32}$$

无向图示例如图 5.1所示。

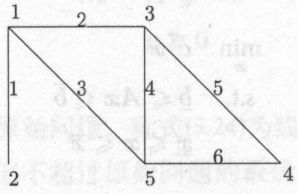


图 5.1 无向图示例

图 5.1所表示的无向图对应的关联矩阵为

$$\begin{pmatrix}1&1&1&0&0&0\\1&0&0&0&0&0\\0&1&0&1&1&0\\0&0&0&0&1&1\\0&0&1&1&0&1\end{pmatrix} \tag{5.33}$$

在无向图的关联矩阵中，每一列恰好包括两个 1，其余为 0。

定义 5.3（有向图的关联矩阵） 有向图 $\boldsymbol{G}(\boldsymbol{V},\boldsymbol{E})$ 的关联矩阵是 $|\boldsymbol{V}|$ 行 $|\boldsymbol{E}|$ 列的矩阵，第 i 行第 j 列的元素 $a_{i,j}$ 满足

$$a_{i,j}=\begin{cases}1, & \text{如果第 } j \text{ 个边指向第 } i \text{ 个顶点}\\ -1, & \text{如果第 } j \text{ 个边离开第 } i \text{ 个顶点}\\ 0, & \text{如果第 } j \text{ 个边与第 } i \text{ 个顶点无关}\end{cases} \tag{5.34}$$

有向图示例如图 5.2所示。

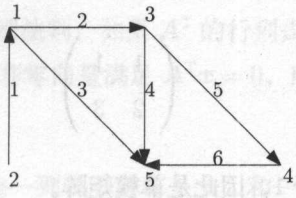


图 5.2 有向图示例

图5.2所表示的有向图对应的关联矩阵为

$$\begin{pmatrix}1 & -1 & -1 & 0 & 0 & 0\\ -1 & 0 & 0 & 0 & 0 & 0\\ 0 & 1 & 0 & -1 & -1 & 0\\ 0 & 0 & 0 & 0 & 1 & -1\\ 0 & 0 & 1 & 1 & 0 & 1\end{pmatrix} \tag{5.35}$$

在有向图的关联矩阵中，每一列恰好包括一个 1 和一个 -1，其余为 0。

5.2.2 全单模矩阵

定义 5.4（单模矩阵，Unimodular Matrix） 单模矩阵是指行列式等于 $+1$ 或 -1 的整数方阵。

- 单位矩阵是单模矩阵。
- 排列矩阵是单模矩阵。
- 单模矩阵的逆矩阵是单模矩阵。
- 单模矩阵的乘积是单模矩阵。

和单模矩阵类似的概念还有全单模矩阵。

定义 5.5（全单模矩阵，Totally Unimodular Matrix） 如果一个整数矩阵的所有子方阵的行列式都等于 0、+1 或 −1，则称其为全单模矩阵。

注意全单模矩阵不再需要是方阵。全单模矩阵中的任何一个元素也可以算作子方阵，因此，根据全单模矩阵的定义，它应当只包含 0、+1 或 −1 这三种元素。另外，这里再给出几个需要避免的误区。

- 单模矩阵可以包含不是 0、+1、−1 的元素。例如

$$\begin{pmatrix} 1 & 1 \\ 2 & 3 \end{pmatrix} \tag{5.36}$$

是整数方阵，且行列式等于 1，因此是单模矩阵。

- 只包含 0、+1 或 −1 的方阵不一定是单模矩阵。例如

$$\begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix} \tag{5.37}$$

是整数方阵，但行列式等于 2，因此不是单模矩阵。

- 只包含 0、+1 或 −1 的矩阵不一定是全单模矩阵。前面的矩阵可以再次用作这里的反例。

5.2.3 全单模矩阵与图论

在本节中我们主要研究两个问题。

- 无向图的关联矩阵何时是全单模矩阵。
- 有向图的关联矩阵何时是全单模矩阵。

通过回答这两个问题，我们可以建立起全单模矩阵与图论之间的关系，为进一步建立线性规划与图论之间的关系做准备。

要证明一个矩阵是全单模矩阵，则需要证明任取子方阵，其行列式都是 $0, 1, -1$。这里我们为了从简单入手，先来讨论一类特殊的行列式为 0 的子方阵。

引理 5.1 任取二分图的关联矩阵的子方阵，如果每列都刚好有两个 1，则其行列式为 0。

证 记 $\boldsymbol{A}$ 为二分图的关联矩阵的子方阵，且 $\boldsymbol{A}$ 每列刚好有两个 1。$\boldsymbol{A}$ 的每一行对应一个顶点，因此可以根据二分图的特点将 $\boldsymbol{A}$ 的行分为两部分。定义向量 $\boldsymbol{x}$ 满足

$$x_i = \begin{cases} 1, & \text{如果}\boldsymbol{A}\text{的第}i\text{行属于第 1 部分} \\ -1, & \text{如果}\boldsymbol{A}\text{的第}i\text{行属于第 2 部分} \end{cases} \tag{5.38}$$

可以验证，$\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = 0$。根据克莱姆法则，如果 $\boldsymbol{A}^{\mathrm{T}}$ 的行列式不是 0，那么满足 $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = 0$ 的 $\boldsymbol{x}$ 只能是零向量。由于我们找到了非零向量满足 $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = 0$，所以 $\boldsymbol{A}^{\mathrm{T}}$ 的行列式为 0，也就是说 $\boldsymbol{A}$ 的行列式为 0。 □

引理 5.2 如果一个矩阵的每一列都刚好有一个 1 和一个 -1，则其行列式为 0。

证 记 $\boldsymbol{A}$ 为有向图的关联矩阵的子方阵，且 $\boldsymbol{A}$ 每列刚好有一个 1 和一个 -1。可以验证，$\boldsymbol{A}^{\mathrm{T}}\boldsymbol{e} = 0$，其中 $\boldsymbol{e}$ 为元素全是 1 的向量。根据克莱姆法则，如果 $\boldsymbol{A}^{\mathrm{T}}$ 的行列式不是 0，那么满足 $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = 0$ 的 $\boldsymbol{x}$ 只能是零向量。由于我们找到了非零向量满足 $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = 0$，所以 $\boldsymbol{A}^{\mathrm{T}}$ 的行列式为 0，也就是说 $\boldsymbol{A}$ 的行列式为 0。 □

在回答无向图的关联矩阵何时是全单模矩阵这一问题之前，我们先证明一个和二分图密切相关的引理。

引理 5.3 无向图当且仅当它不存在奇环（总边数为奇数的环）时是二分图。

证 二分图不存在奇环是显然的。我们知道在二分图中有两个顶点集合，从任一点出发经过奇数条边所到达的顶点与出发点必然不在同一顶点集合中，因此就不可能形成环。

如果在一个图中不存在奇环，则我们来证明它一定是二分图。我们不妨假设图是连通的，如果不连通则可以对每个连通分量分别做类似的推导。任选一个顶点 $\boldsymbol{v}$，再假设每条边的长度都是 1，我们可以根据各个顶点到 $\boldsymbol{v}$ 的最短距离是奇数还是偶数将其划分至两个集合。如果两个顶点到 $\boldsymbol{v}$ 的最短距离都是奇数，那么这两个顶点之间不可能有边，否则就会形成奇环。同理，如果两个顶点到 $\boldsymbol{v}$ 的最短距离都是偶数，那么这两个顶点之间也不可能有边。因此同一集合中的顶点之间不存在边，也就证明了这是一个二分图。 □

现在我们可以给出定理，阐明无向图的关联矩阵是全单模矩阵的充分必要条件。

定理 5.3 无向图的关联矩阵当且仅当是二分图时是全单模矩阵。

证 我们首先证明：如果无向图不是二分图，则其关联矩阵不是全单模矩阵。根据引理5.3，如果无向图不是二分图，那么它就有奇环。因此，通过调整行列顺序，可以选出奇环对应的子方阵为：

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & \cdots & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 1 & 1 \end{pmatrix} \tag{5.39}$$

这个子方阵是奇数阶方阵，根据第 1 列进行展开，容易算得其行列式为 2。因此关联矩阵不是全单模矩阵。

下面我们证明：如果无向图是二分图，则其关联矩阵是全单模矩阵。我们使用数学归纳法，如果选择关联矩阵的 1 阶子方阵，也就是一个数，则只能是 0、1 或 -1，所以行列式也只能等于 0、1 或 -1。假设关联矩阵的 k 阶子方阵的行列式只能是 0、1 或 -1，则我们来证明 $k+1$ 阶子方阵有同样的性质。我们根据 $k+1$ 阶子方阵各列包含的非零的个数分类讨论。

- 如果 $k+1$ 阶子方阵的所有列都有两个 1，则根据引理5.1，其行列式为 0。
- 如果 $k+1$ 阶子方阵存在一列全是 0，则其行列式为 0。
- 如果 $k+1$ 阶子方阵存在一列刚好包含一个 1，则利用这个 1 进行行列式展开，可知 $k+1$ 阶子方阵的行列式等于 k 阶子方阵的行列式或其相反数。由归纳假设知 $k+1$ 阶子方阵的行列式只能是 0、1 或 -1。

综上所述，无向图的关联矩阵是全单模矩阵当且仅当是二分图。 □

对于有向图，我们的结论比无向图更优美。

定理 5.4 有向图的关联矩阵是全单模矩阵。

证 我们使用数学归纳法，如果选择关联矩阵的 1 阶子方阵，也就是一个数，则只能是 0、1 或 -1，所以行列式也只能等于 0、1 或 -1。假设关联矩阵的 k 阶子方阵的行列式只能

是 0、1 或 -1，则我们来证明 $k+1$ 阶子方阵有同样的性质。我们根据 $k+1$ 阶子方阵各列包含的非零的个数分类讨论。

- 如果 $k+1$ 阶子方阵的所有列都有一个 1、一个 -1，则根据引理5.2，其行列式为 0。
- 如果 $k+1$ 阶子方阵存在一列全是 0，则其行列式为 0。
- 如果 $k+1$ 阶子方阵存在一列仅包含一个非零元素，则利用这个非零元素进行行列式展开，可知 $k+1$ 阶子方阵的行列式等于 k 阶子方阵的行列式或其相反数。由归纳假设知 $k+1$ 阶子方阵的行列式只能是 0、1 或 -1。

综上所述，有向图的关联矩阵是全单模矩阵。 □

5.2.4 全单模矩阵与线性规划

如果在线性规划中包含全单模矩阵，则会有非常有意思的结果。我们考虑线性规划

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \geqslant 0 \end{aligned} \tag{5.40}$$

则对于任意一个可行基 $\mathcal{B}$，我们有 $\boldsymbol{A}_{\mathcal{B}}\boldsymbol{x}_{\mathcal{B}} = \boldsymbol{b}$。如果 $\boldsymbol{A}$ 是全单模矩阵，且 $\boldsymbol{b}$ 中的元素都是整数，则 $\det(\boldsymbol{A}_{\mathcal{B}})$ 为 0，1 或者 -1。根据克莱姆法则，方程 $\boldsymbol{A}_{\mathcal{B}}\boldsymbol{x}_{\mathcal{B}} = \boldsymbol{b}$ 的解一定是整数，因此线性规划存在整数最优解。这就说明对于整数规划问题

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \in \mathcal{Z}^{+} \end{aligned} \tag{5.41}$$

我们可以去掉 $\boldsymbol{x}$ 必须是整数的限制。因为即使没有这个限制，问题同样存在整数最优解。

我们还要说明，即使约束不是等式，这一结论也是成立的。考虑线性规划

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \geqslant \boldsymbol{b} \\ & \boldsymbol{x} \geqslant 0 \end{aligned} \tag{5.42}$$

我们记

$$\boldsymbol{A}' = (\boldsymbol{A}, -\boldsymbol{I}), \boldsymbol{x}' = \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{pmatrix}, \boldsymbol{c}' = \begin{pmatrix} \boldsymbol{c} \\ 0 \end{pmatrix} \tag{5.43}$$

则线性规划问题等价于

$$\begin{aligned} \min_{\boldsymbol{x}'} \quad & \boldsymbol{c}'^{\mathrm{T}} \boldsymbol{x}' \\ \text{s.t.} \quad & \boldsymbol{A}' \boldsymbol{x}' = \boldsymbol{b} \\ & \boldsymbol{x} \geqslant 0 \end{aligned} \tag{5.44}$$

如果矩阵 $\boldsymbol{A}$ 是全单模矩阵，则我们可以证明 $\boldsymbol{A}' = (\boldsymbol{A}, \boldsymbol{I})$ 也是全单模矩阵。这是因为任取 $\boldsymbol{A}'$ 的子矩阵，如果其中包含 $\boldsymbol{I}$ 中的列，则该列至多只有一个 1，其余全是 0。如果不包含 1，则行列式为 0，否则我们可以对这种列进行行列式展开。不断重复这样的操作直到剩下的子矩阵仅包含 $\boldsymbol{A}$ 中的列。这时因为 $\boldsymbol{A}$ 是全单模矩阵，所以最终行列式只能是 $0, 1, -1$。因此这一问题也存在整数最优解。

全单模矩阵的性质告诉了我们一些很有用的信息。涉及二分图和有向图的许多图论问题，都是容易求解的，这是因为二分图与有向图的关联矩阵都是全单模矩阵。这时整数规划中的整数约束可以去掉，变为普通的线性规划，也就是多项式时间可解的。相反，涉及一般无向图的许多图论问题，都是难于求解的，这是因为一般无向图的关联矩阵不是全单模矩阵。如果整数规划没有其他特殊的性质，则通常是 $\mathcal{NP}$-难问题。虽然这个结论并不绝对，但确实具有一定的普遍意义。

5.3 图论中的经典问题

在本节中，我们将列举一些图论中的经典问题，并用线性规划去描述，这说明线性规划可以用于求解图论中的问题。另外，我们也将使用全单模矩阵、对偶等理论去证明图论中的一些重要结论。

5.3.1 单源最短路问题

单源最短路是指从图中的某一指定点出发，到达其他所有点的最短路径。记图 $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ 中，边 (i, j) 的长度为 $w_{i,j}$。我们将从两个角度将单源最短路问题写成最优化问题：对每一条边设一个变量，或对每一个点设一个变量。

我们先来考虑第 1 个角度。对每一条边 (u,v)，我们记变量 $x_{u,v}$ 满足

$$x_{u,v}=\begin{cases}1, & \text{最短路经过边}(u,v)\\ 0, & \text{最短路不经过边}(u,v)\end{cases} \tag{5.45}$$

对每一个点 i，我们考虑所有关联的边的对应变量和，即 $\sum\limits_j x_{j,i}-\sum\limits_j x_{i,j}$。根据 i 的不同，它有三种不同的取值。

$$\sum_{(j,i)\in \boldsymbol{E}} x_{j,i}-\sum_{(i,j)\in \boldsymbol{E}} x_{i,j}=\begin{cases}-1, & i=s\\ 1, & i=t\\ 0, & i\neq s,t\end{cases} \tag{5.46}$$

这是因为 $i=s$ 时只会有一条出边，$i=t$ 时只会有一条入边，而其他情况或者出边、入边各有一条，或者无任何边。

因此我们可以得到这样一个最优化问题。

$$\begin{aligned}\min_{\boldsymbol{x}} \quad & \sum_{(i,j)\in \boldsymbol{E}} w_{i,j}x_{i,j}\\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x}=\boldsymbol{e}_t-\boldsymbol{e}_s\\ & \boldsymbol{x}\in\{0,1\}\end{aligned} \tag{5.47}$$

其中 $\boldsymbol{A}$ 是图的关联矩阵，$\boldsymbol{e}_t$ 表示第 t 行等于 1、其余行都等于 0 的列向量。类似地，$\boldsymbol{e}_s$ 表示第 s 行等于 1、其余行都等于 0 的列向量。

我们可以证明，问题(5.47)中的 $\boldsymbol{x}\in\{0,1\}$ 可以改为 $\boldsymbol{x}\geqslant 0$，即等价于

$$\begin{aligned}\min_{\boldsymbol{x}} \quad & \sum_{(i,j)\in \boldsymbol{E}} w_{i,j}x_{i,j}\\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x}=\boldsymbol{e}_t-\boldsymbol{e}_s\\ & \boldsymbol{x}\geqslant 0\end{aligned} \tag{5.48}$$

定理 5.5 问题(5.47)与问题(5.48)等价。

因为有向图的关联矩阵是全单模矩阵，且增广矩阵 $(\boldsymbol{A},\boldsymbol{e}_t-\boldsymbol{e}_s)$ 也是全单模矩阵，所以根据克莱姆法则，无论如何选取基，最终 $\boldsymbol{x}$ 的各个分量都只会等于 0 或 1。

另外一种将单源最短路问题写成最优化问题的思路，是对每一个点设一个变量。我们记 y_i 表示从 s 出发到 i 的最短距离。对于每一条边 (u,v)，我们有 $y_v\leqslant y_u+w_{u,v}$。这是因

为我们可以先从 s 到 u，然后继续沿着边 (u,v) 走到 v。这样的总距离至少是 $y_u + w_{u,v}$，应当大于或等于 s 到 v 的最短距离。我们的目标是要 $y_t - y_s$ 达到最大。可以写成如下最优化问题。

$$\begin{aligned} \max_{\boldsymbol{y}} \quad & y_t - y_s \\ \text{s.t.} \quad & \boldsymbol{A}^{\mathrm{T}}\boldsymbol{y} \leqslant w \end{aligned} \tag{5.49}$$

我们可以验证问题(5.48)和问题(5.49)互为对偶问题。

5.3.2 二分图的最大匹配与最小覆盖问题

二分图的最大匹配与最小覆盖是图论中的经典问题。本节我们利用线性规划的对偶来理解这两个问题。

定义 5.6（匹配） 给定一个无向图 $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$，$\boldsymbol{G}$ 的匹配是指其若干不相邻边的集合，也就是说如果 $\boldsymbol{M} \subseteq \boldsymbol{E}$ 是 $\boldsymbol{G}$ 的匹配，则 $\boldsymbol{M}$ 中的任意两条边都没有公共顶点。

定义 5.7（最大匹配） 给定一个无向图 $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$，如果匹配 $\boldsymbol{M}$ 在 $\boldsymbol{G}$ 的所有匹配中边数最多，则称其为最大匹配。

下面我们考虑将二分图的最大匹配写为线性规划。对每一条边 e，我们定义一个变量 $x_e \in \{0,1\}$，当且仅当 $x_e = 1$ 时我们将 e 加入匹配。根据匹配的定义，每一个顶点关联的所有边中，应当至多只有一条边加入匹配。这也就是说，对于任意的顶点 v，我们有 $\sum\limits_{ev} x_e \leqslant 1$。我们要求最大匹配，也就是希望 $\sum\limits_{e \in \boldsymbol{E}} x_e$ 达到最大。综上所述，二分图的最大匹配可以写成如下最优化问题。

$$\begin{aligned} \max_{\boldsymbol{x}} \quad & \sum_{e \in \boldsymbol{E}} x_e \\ \text{s.t.} \quad & \sum_{ev} x_e \leqslant 1, \forall v \in \boldsymbol{V} \\ & x_e \in \{0,1\}, \forall e \in \boldsymbol{E} \end{aligned} \tag{5.50}$$

其中 $x_e \in \{0,1\}$ 这个条件可以放宽为 $x_e \geqslant 0$。由于二分图的关联矩阵是全单模矩阵，所以即使只要求 $x_e \geqslant 0$，也存在整数最优解。而 x_e 非负，所以 $\sum\limits_{ev} x_e \leqslant 1$ 实际上已经潜在要求了 $x_e \leqslant 1$。因此最优化问题(5.50)可以改为

$$\begin{aligned} \max_{x} \quad & \sum_{e \in \boldsymbol{E}} x_e \\ \text{s.t.} \quad & \sum_{ev} x_e \leqslant 1, \forall v \in \boldsymbol{V} \\ & x_e \geqslant 0, \forall e \in \boldsymbol{E} \end{aligned} \tag{5.51}$$

定义 5.8（顶点覆盖） 给定一个无向图 $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$，我们称子集 $\boldsymbol{C} \subseteq \boldsymbol{V}$ 是 $\boldsymbol{G}$ 的顶点覆盖，当且仅当每一条边都至少有一个端点属于 $\boldsymbol{C}$。

定义 5.9（最小顶点覆盖） 给定一个无向图 $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$，如果顶点覆盖 $\boldsymbol{C}$ 在 $\boldsymbol{G}$ 的所有顶点覆盖当中顶点数最少，则称其为最小顶点覆盖。

类似地，我们也可以定义图的边覆盖、最小边覆盖。在本节中我们只考虑顶点覆盖和最小顶点覆盖，因此简称为覆盖和最小覆盖。下面我们考虑将二分图的最小覆盖写为线性规划。对每一个顶点 v，我们定义一个变量 $x_v \in \{0, 1\}$。当且仅当 $x_v = 1$ 时我们将 v 加入覆盖。根据覆盖的定义，每一条边都至少有一个端点属于覆盖。这也就是说，对于任意边 e，我们有 $\sum_{ve} x_v \geqslant 1$。我们要求最小覆盖，也就是希望 $\sum_{v \in \boldsymbol{V}} x_v$ 达到最小。综上所述，二分图的最小覆盖可以写成如下最优化问题。

$$\begin{aligned} \min_{x} \quad & \sum_{v \in \boldsymbol{V}} x_v \\ \text{s.t.} \quad & \sum_{ve} x_v \geqslant 1, \forall e \in \boldsymbol{E} \\ & x_v \in \{0, 1\}, \forall v \in \boldsymbol{V} \end{aligned} \tag{5.52}$$

类似前面对最大匹配的分析，我们同样可以将 $x_v \in \{0, 1\}$ 这个条件放宽为 $x_v \geqslant 0$。因此最优化问题 (5.52) 可以改为

$$\begin{aligned} \min_{x} \quad & \sum_{v \in \boldsymbol{V}} x_v \\ \text{s.t.} \quad & \sum_{ve} x_v \geqslant 1, \forall e \in \boldsymbol{E} \\ & x_v \geqslant 0, \forall v \in \boldsymbol{V} \end{aligned} \tag{5.53}$$

通过简单的验证可以发现，问题 (5.51) 和问题 (5.53) 互为对偶。于是我们得到了 König 定理，即定理 5.6。

定理 5.6 二分图的最大匹配数等于最小覆盖数。

5.3.3 最大流与最小割问题

最大流与最小割也是图论中的重要问题。

定义 5.10（最大流问题，Max Flow Problem） 给定一个有向图 $\boldsymbol{G}=(\boldsymbol{V},\boldsymbol{E})$，图中包含一个源点 $s\in\boldsymbol{V}$ 和一个汇点 $t\in\boldsymbol{V}$。图中的每条有向边都有一定的流量限制。最大流问题就是要求在流量限制下，由 s 通过若干中间节点向 t 发送总量尽可能大的流。

为了表述方便，我们可以认为添加一条由 t 指向 s 的有向边，流量限制为无穷大，并定义 $\boldsymbol{E}'=\boldsymbol{E}\cup\{(t,s)\}$。这样流就可以从 t 回到 s，于是最大流问题变为寻找尽可能大的环流。我们记 $c_{i,j}$ 表示边 (i,j) 上的流量限制，其中 $c_{t,s}=+\infty$，再记变量 $f_{i,j}$ 表示边 (i,j) 上的流量，则最大流问题可以表述如下。

$$\begin{aligned}\max_{\boldsymbol{f}}\quad & f_{t,s}\\ \text{s.t.}\quad & \sum_{(j,i)\in\boldsymbol{E}'}f_{j,i}-\sum_{(i,j)\in\boldsymbol{E}'}f_{i,j}=0, && \forall i\in\boldsymbol{V}\\ & 0\leqslant f_{i,j}\leqslant c_{i,j}, && \forall(i,j)\in\boldsymbol{E}'\end{aligned}\tag{5.54}$$

我们可以写出的最大流问题的对偶问题如下。

$$\begin{aligned}\min_{\boldsymbol{y},\boldsymbol{z}}\quad & \sum_{(i,j)\in\boldsymbol{E}}c_{i,j}y_{i,j}\\ \text{s.t.}\quad & z_j-z_i\leqslant y_{i,j}, && \forall(i,j)\in\boldsymbol{E}\\ & z_t-z_s\geqslant 1,\\ & y_{i,j}\geqslant 0, && \forall(i,j)\in\boldsymbol{E}\end{aligned}\tag{5.55}$$

我们试图直接去理解式 (5.55) 的含义，可以根据 z 将顶点划分为两个集合。

$$\boldsymbol{U}=\{v\in\boldsymbol{V}\mid z_v<z_t\}\tag{5.56}$$

$$\bar{\boldsymbol{U}}=\boldsymbol{V}-\boldsymbol{U}=\{v\in\boldsymbol{V}\mid z_v\geqslant z_t\}\tag{5.57}$$

如果边 $(i,j)\in\boldsymbol{E}$，且 $i\in\boldsymbol{U}$、$j\in\bar{\boldsymbol{U}}$，则

$$y_{i,j}\geqslant z_j-z_i\geqslant z_t-z_s\geqslant 1\tag{5.58}$$

所以

$$\sum_{(i,j)\in \boldsymbol{E}} c_{i,j}y_{i,j} \geqslant \sum_{(i,j)\in \boldsymbol{E}, i\in \boldsymbol{U}, j\in \bar{\boldsymbol{U}}} c_{i,j}y_{i,j} \geqslant \sum_{(i,j)\in \boldsymbol{E}, i\in \boldsymbol{U}, j\in \bar{\boldsymbol{U}}} c_{i,j} \tag{5.59}$$

我们可以构造一个可行解

$$\hat{y}_{i,j} = \begin{cases} 1, & (i,j) \in \boldsymbol{E}, i \in \boldsymbol{U}, j \in \bar{\boldsymbol{U}} \\ 0, & \text{其他} \end{cases} \tag{5.60}$$

这容易验证 $\hat{\boldsymbol{y}}$ 的可行性，并且 $\hat{\boldsymbol{y}}$ 可以使得式 (5.59) 取到等号，也就说明它是一个最优解。所以式 (5.55) 可以看作求解

$$\begin{aligned} \min_{\boldsymbol{U}} \quad & \sum_{(i,j)\in \boldsymbol{E}, i\in \boldsymbol{U}, j\in \boldsymbol{V}-\boldsymbol{U}} c_{i,j} \\ \text{s.t.} \quad & s \in \boldsymbol{U}, t \in \boldsymbol{V} - \boldsymbol{U} \end{aligned} \tag{5.61}$$

而式 (5.61) 正是最小割问题，这也就证明了最大流-最小割定理。

定理 5.7（最大流-最小割定理） 最大流问题与最小割问题互为对偶，且最大流等于最小割。

5.4 延伸阅读

在实际问题中，图论相关的问题往往更为复杂，无法写成线性规划。但其中有一些问题仍然与线性规划有相似之处，值得借鉴。

5.4.1 逐步线性规划

逐步线性规划（Successive Linear Programming）是一种求解非线性规划的方法。它的基本思想就是通过多次求解与原问题相似的线性规划，最终得到原问题的解。逐步线性规划在混合物流问题上有很好的应用。

混合物流问题（Pooling Problem）是石油化工生产中的一类问题。在石油化工生产中，经常遇到混合物流问题，例如不同性质的原油混炼、产品调和等。在混合物流的计划模型中，人们关注如何正确计算混合物流的性质。其中的某些性质如辛烷值、硫化物含量等对于不同的产品均有不同的要求。

混合物流问题的基本模型与最大流问题非常相似，但在化工生产当中，我们还需要考虑流的多种化学性质。这个问题的数学模型最早是由 [Hav78] 提出的。如图5.3所示，原油A、B、C经过一系列混合，最终得到两种产品X、Y。表5.1给出了与其对应的数据。

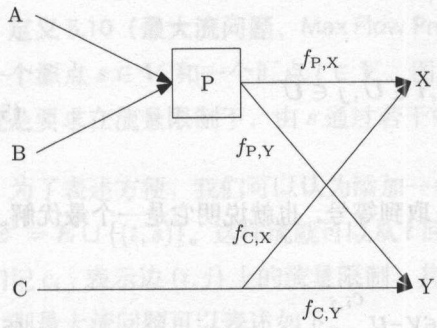


图 5.3　混合物流问题示例

表 5.1　混合物流问题的数据示例

编　号	硫化物含量	单　价	总　量
A	3%	\$6	$[0,+\infty)$
B	1%	\$16	$[0,+\infty)$
C	2%	\$10	$[0,+\infty)$
X	$[0\%,2.5\%]$	\$9	$[0,100]$
Y	$[0\%,1.5\%]$	\$15	$[0,200]$

类似最大流问题，我们可以建立最优化模型。但与最大流问题不同的是，这里还需要考虑硫化物的含量，这就使得我们需要考虑一类非线性的约束条件。也就是说，除了传统的流量守恒约束，我们还需要满足硫化物含量的质量守恒定律。我们用 $f_{i,j}$ 表示点 i 到点 j 的流量，用 w_i 表示点 i 处的硫化物含量。图5.3与表5.1所对应的最优化问题可以写成如下形式。

$$
\begin{aligned}
\min_{f,w}\quad & 6f_{A,P}+16f_{B,P}+10f_{C,X}+10f_{C,Y} \\
& -9f_{P,X}-9f_{C,X}-15f_{P,Y}-15f_{C,Y} \\
\text{s.t.}\quad & f_{P,X}+f_{C,X}\leqslant 100 \\
& f_{P,Y}+f_{C,Y}\leqslant 200 \\
& f_{A,P}+f_{B,P}-f_{P,X}-f_{P,Y}=0 \\
& w_P(f_{P,X}+f_{P,Y})-f_{P,X}-f_{P,Y}=0 \\
& w_P f_{P,X}+2f_{C,X}-2.5(f_{P,X}+f_{C,X})\leqslant 0 \\
& w_P f_{P,Y}+2f_{C,Y}-1.5(f_{P,X}+f_{C,X})\leqslant 0 \\
& f_{A,P},f_{B,P},f_{P,X},f_{P,Y},f_{C,X},f_{C,Y}\geqslant 0
\end{aligned}
\tag{5.62}
$$

其中目标函数是最小化支出与收入之差，仍然是线性函数，但约束已不再是简单的线性函数。[AH13] 利用最大独立集问题证明了混合物流问题是强 $\mathcal{NP}$-难的，这也说明了混合物流问题同线性规划的本质区别。但由于它们之间在形式上非常相似，所以在设计算法时还是可以借鉴线性规划的思想的。例如 [BL85, LWSP79] 都使用了逐步线性规划方法进行求解。

5.4.2 半正定规划

半正定规划（Semidefinite Programming）可以认为是线性规划的一种推广。半正定规划中的变量不再是向量，而是半正定矩阵。半正定规划由于在最大割问题上的成功应用[GW95]而获得更多关注。

定义 5.11（最大割问题，Maximum Cut Problem） 将无向图中的顶点划分为两个集合，使得不同集合之间的边权和最大。

假设我们有 n 个顶点，则对每一个顶点 i，我们定义一个变量 x_i 来表示它属于哪个集合。其中一个集合中的顶点满足 $x_i = 1$，另一个集合中的顶点满足 $x_i = -1$。因此对于两个顶点 i 和 j，当且仅当边 (i,j) 在不同集合之间时 $x_i x_j = -1$，当且仅当边 (i,j) 在同一集合之中时 $x_i x_j = 1$。我们记矩阵 $\boldsymbol{W}$ 是图的边权矩阵，即当存在边 (i,j) 时，$W_{i,j}$ 为该边的权重，当不存在边 (i,j) 时，$W_{i,j} = 0$。从而最大割问题可以写为

$$\begin{aligned} \max_{\boldsymbol{x}} \quad & \frac{1}{2} \sum_{1 \leqslant i < j \leqslant n} (1 - x_i x_j) W_{i,j} \\ \text{s.t.} \quad & \boldsymbol{x} \in \{-1, 1\} \end{aligned} \tag{5.63}$$

下面我们定义一个矩阵 $\boldsymbol{X} = \boldsymbol{x}\boldsymbol{x}^{\mathrm{T}}$，则 $\boldsymbol{X}$ 有这样几个特点。

- $\boldsymbol{X}$ 是半正定矩阵。
- $X_{i,j} = x_i x_j$。
- $X_{i,i} = x_i^2 = 1$。

最大割问题可以等价地改写为

$$\begin{aligned} \min_{\boldsymbol{x}, \boldsymbol{X}} \quad & < \boldsymbol{W}, \boldsymbol{X} > \\ \text{s.t.} \quad & \boldsymbol{x} \in \{-1, 1\} \\ & \boldsymbol{X} = \boldsymbol{x}\boldsymbol{x}^{\mathrm{T}} \end{aligned} \tag{5.64}$$

其中 $< \boldsymbol{W}, \boldsymbol{X} >$ 代表矩阵内积，即两个矩阵对应位置的元素的乘积之和。

$$< \boldsymbol{W}, \boldsymbol{X} > = \sum_{i=1}^{n} \sum_{j=1}^{n} W_{i,j} X_{i,j} \tag{5.65}$$

下面我们做一个不等价的变换。我们只要求 $\boldsymbol{X}$ 是半正定矩阵，对角线元素全都是 1，不再要求 $\boldsymbol{X}=\boldsymbol{x}\boldsymbol{x}^{\mathrm{T}}$。于是问题变为

$$\begin{aligned}\min_{\boldsymbol{X}} \quad & <\boldsymbol{W},\boldsymbol{X}> \\ \text{s.t.} \quad & X_{i,i}=1,\forall i=1,2,\cdots,n \\ & \boldsymbol{X}\succeq 0\end{aligned} \tag{5.66}$$

其中，约束 $\boldsymbol{X}\succeq 0$ 指 $\boldsymbol{X}$ 是半正定矩阵。在这个问题中，变量 $\boldsymbol{X}$ 是半正定矩阵，因此被称为半正定规划。通过求解半正定规划并进行一些后处理，得到的近似解可以达到 0.878 的近似比。

第6章

无约束优化

有一类非常广泛的研究领域叫作最优化方法，本章要研究的内容属于其中一部分。我们在本章研究的问题在整个最优化方法当中相对简单，仅限制在无约束最优化方法这一个方面。用数学的语言来说，就是对于一个目标函数 $f(x)$，我们如何尽可能地找到它的最优值(最大值或最小值)。这是在统计、物理、金融、计算机等领域中都会涉及的一类非常基本的问题。针对这样一个问题，我们需要抓住 $f(x)$ 的特点进行具体分析，选用最为合适的算法。

最优值通常是不容易找的，所以有的时候我们会降低要求，这就会涉及最值点、极值点和稳定点。因此在介绍具体算法之前，我们先来认识一下这三者之间的区别。最值点是指让函数在整个定义域上取得最小值的点，也就是我们在理想情况下要找到的那个点；极值点是指让函数在某个邻域内取得最小值的点，也就是说它虽然不一定是最值点，但从小范围来看也算是最优的了；稳定点是指让函数梯度为零的点，它有可能是极值点，但也不一定。这几个概念很容易混淆。我们的目标是希望得到最值点，即可以使 $f(x)$ 达到最小的 x。这样通常比较困难，所以很多时候能够求得极值点也算非常不错了。但许多算法实际上也不能保证求得极值点，而只是保证求得稳定点，这是因为验证一个点比它附近的所有点都好并不是一件容易的事情。事实上本章的许多方法都只能保证求得稳定点，而不能求得极值点。一个可以经常用到的反例是 $f(x)=x^3$，它在 $x=0$ 的时候梯度为 0，因此如果从这个点开始迭代，则许多使用梯度的算法都是直接终止的。然而 $x=0$ 明显不是极值点，更不是最值点。

在本章中我们首先介绍单峰函数这样一类特殊函数的优化方法；然后根据是否需要用到导数，分别介绍几种无导数优化方法和导数优化方法；最后针对最小二乘这样一个非常

经典的无约束优化问题介绍一些算法。

6.1 单峰函数的最值

定义 6.1（单峰函数，Unimodal Function） 对于函数 $f(x)$，若存在一个值 m，当 $x \leqslant m$ 时函数严格单调递增，$x \geqslant m$ 时函数严格单调递减，则 $f(x)$ 叫作单峰函数。

我们列举一些简单的单峰函数。

- $f(x) = ax^2 + bx + c(a < 0)$ 是单峰函数，分界点 $m = -\dfrac{b}{2a}$。
- $f(x) = 2^{-x^2}$ 是单峰函数，分界点 $m = 0$。
- $f(x) = -|x - m|$ 是单峰函数。
- $f(x) = \cos x$ 在 $[-\pi, \pi]$ 上是单峰函数，分界点 $m = 0$。

在其中第 3 个例子中考虑的是一个区间上的单峰函数。在实际问题中，通常只考虑一个区间 $[L, U]$ 上的单峰函数，求解 $f(x)$ 的分界点，即 m。这是因为在计算上我们关心的 x 的取值范围总是有限的，如果有必要，则我们只需将区间范围选到足够大。而实际问题中的单峰函数 $f(x)$ 不会像我们列举的几个函数这样简单，只是特定领域的先验知识能够让我们确信它是一个单峰函数。因此单纯靠具体的 $f(x)$ 的表达式是不足以推导出 m 的，只能通过某种计算方法来找。

在本节中将要介绍的方法都基于类似的思路：逐渐缩小所需考虑的区间。刚开始我们需要考虑的区间长度是 $U - L$，逐渐排除不包含最优解的部分，最终得到一个长度小于 ϵ 的区间。排除的方法则主要依靠定理6.1。

定理 6.1 假设有两个点 l, u，满足 $L \leqslant l < u \leqslant U$，记单峰函数的最优解是 m，则我们有：

- 如果 $f(l) < f(u)$，则 $m \in [l, U]$；
- 如果 $f(l) > f(u)$，则 $m \in [L, u]$；
- 如果 $f(l) = f(u)$，则 $m \in (l, u)$。

定理6.1可以用反证法来证明：如果 m 在其余范围内，则都会导致矛盾。这一定理的意义在于，我们可以通过计算两个点的函数值来缩小区间范围，逐渐找到 m。那进一步的问题就是：如何选择 l 和 u。下面我们讨论几种具体的方法，并对比它们的优劣。

6.1.1 三分法

三分法（Ternary Search）是解决单峰函数最值问题的最常见的方法之一。三分法的思想很简单：直接将 l 和 u 选取为三等分点。虽然不一定是最好的选择方法，但确实是足够简单的选择方法。

算法 37 三分法

输入： 搜索区间 $[L,U]$，精度 $\epsilon > 0$

1: $l \leftarrow L, u \leftarrow U$。
2: **while** $u - l \geqslant \epsilon$ **do**
3: $p_l \leftarrow (2l+u)/3$，$p_u \leftarrow (l+2u)/3$。
4: **if** $f(p_l) < f(p_u)$ **then**
5: $l \leftarrow p_l$。
6: **else**
7: $u \leftarrow p_u$。
8: **end if**
9: **end while**

我们可以估计一下最坏情形下的计算量。实际上对于三分法，不存在最坏情况，我们总是只有一种情况。每一次循环，我们都会计算 2 次函数值，使区间长度变为原来的 $2/3$。因此计算 n 次函数值后，我们可以使区间长度变为原来的 $(2/3)^{n/2}$。如果需要区间长度小于 ϵ，则函数值计算次数应当满足 $(2/3)^{n/2} < \epsilon$，即 $n > 2\log_{2/3}\epsilon$。也就是说，函数值的计算次数超过 $2\log_{2/3}\epsilon$ 次以后，我们可以使区间长度缩小至 ϵ 以下。后面我们可以通过比较这个值来对比算法的优劣。

6.1.2 对分法

在三分法中，我们选取的是两个三等分点。如果尝试让这两个点靠近中点，则会发现计算速度有所提高。在极限情况下就是让它们都非常靠近中点，也就是对分法。

算法 38 对分法

输入： 搜索区间 $[L,U]$，精度 $\epsilon > 0$，参数 $0 < \delta < \epsilon/2$

1: $l \leftarrow L, u \leftarrow U$。
2: **while** $u - l \geqslant \epsilon$ **do**

3: $p_l \leftarrow (l+u)/2-\delta$，$p_u \leftarrow (l+u)/2+\delta$。
4: **if** $f(p_l) < f(p_u)$ **then**
5: $l \leftarrow p_l$。
6: **else**
7: $u \leftarrow p_u$。
8: **end if**
9: **end while**

我们每计算两次函数值，都可以使区间长度大约变为原来的 $1/2$。因此计算 n 次函数值后，我们可以使区间长度变为原来的 $(1/2)^{n/2}$。如果需要区间长度小于 ϵ，则函数值计算次数应当满足 $(1/2)^{n/2} < \epsilon$，即 $n > 2\log_{1/2}\epsilon$。由于 $\log_{1/2}\epsilon < \log_{2/3}\epsilon$，所以对分法比三分法的速度更快。更进一步，我们可以算出对分法所需的时间是三分法的 $\dfrac{\ln(2/3)}{\ln(1/2)} \approx 0.585$ 倍。

6.1.3 黄金分割法

在之前的两个方法中，每次迭代都要计算两次函数值。但实际上前一次迭代时计算函数值的两个点之中，会有一个点仍然在当前的搜索区间内。因此，我们可以再次使用上一次迭代时计算的函数值，再另外计算一次函数值，这样就足以用于缩小搜索区间了。

假设当前搜索区间是 $[l,u]$，我们计算两个关于 $\dfrac{l+u}{2}$ 对称的点 $p_l = \phi l + (1-\phi)u$ 和 $p_u = (1-\phi)l + \phi u$ 的函数值，其中 ϕ 是待定参数。为了让 $l < p_l < p_u < u$，应当有 $\phi \in (\frac{1}{2}, 1)$。不妨假设 $f(p_l) < f(p_u)$，则搜索区间缩小为 $[p_l, u]$。在下一步的迭代中，我们要计算的两个关于 $\dfrac{l+u}{2}$ 对称的点应当是 $\phi p_l + (1-\phi)u$ 和 $(1-\phi)p_l + \phi u$。为了能够再次利用到之前计算过的函数值 p_u，我们希望这两个点中有一个等于 p_u，所以 $\phi p_l + (1-\phi)u = p_u$ 或者 $(1-\phi)p_l + \phi u = p_u$。

- 如果 $\phi p_l + (1-\phi)u = p_u$，则我们将 p_l 和 p_u 的表达式代入，化简后得到 $(\phi^2+\phi-1)(u-l)=0$，因此只能是 $\phi^2+\phi-1=0$。由于 $\phi \in (\frac{1}{2}, 1)$，所以解得 $\phi = \dfrac{\sqrt{5}-1}{2}$。
- 如果 $(1-\phi)p_l + \phi u = p_u$，则我们将 p_l 和 p_u 的表达式代入，化简后得到 $(1-\phi)^2(u-l) = 0$。由于 $\phi \in (\frac{1}{2}, 1)$，所以无解。

为了满足所期望的要求，我们应当选择 $\phi = \dfrac{\sqrt{5}-1}{2}$，而这刚好就是黄金分割比。所以这种方法叫作黄金分割法。

算法 39　黄金分割法

输入： 搜索区间 $[L,U]$，精度 $\epsilon>0$。

$\phi \leftarrow \dfrac{\sqrt{5}-1}{2}$。
$l \leftarrow L$，$u \leftarrow U$。
$p_l \leftarrow \phi l+(1-\phi)u$，$p_u \leftarrow (1-\phi)l+\phi u$。
$f_l \leftarrow f(p_l)$，$f_u \leftarrow f(p_u)$。
while $u-l \geqslant \epsilon$ **do**
  if $p_l < p_u$ **then**
    $l \leftarrow p_l$。
    $p_l \leftarrow p_u$，$p_u \leftarrow (1-\phi)l+\phi u$。
    $f_l \leftarrow f_u$，$f_u \leftarrow f(p_u)$。
  else
    $u \leftarrow p_u$。
    $p_u \leftarrow p_l$，$p_l \leftarrow \phi l+(1-\phi)u$。
    $f_u \leftarrow f_l$，$f_l \leftarrow f(p_l)$。
  end if
end while

除了前两次的函数值计算，每计算 1 次函数值就可以将区间长度变为原来的 $\dfrac{\sqrt{5}-1}{2}\approx 0.618$ 倍。如果我们忽略最初的两次函数值计算，则 n 次函数值计算可以使区间长度变为原来的 0.618^n 倍。如果需要区间长度小于 ϵ，则函数值计算次数应当满足 $0.618^n<\epsilon$，即 $n>\log_{0.618}\epsilon$。因此，黄金分割法所需的时间是对分法的 $\dfrac{\ln(1/2)}{2\ln(0.618)}\approx 0.720$ 倍，是三分法的 $\dfrac{\ln(2/3)}{2\ln(0.618)}$ 即约 0.421 倍。

6.1.4　小结

如果直接把本节的几种方法应用到一元多峰函数效果会如何？实际问题往往无法保证是严格单峰函数，对于一些问题，如果不是单峰函数，则其实也可以尝试用上述方法去解。如果函数是双峰、三峰等，则我们仍然会找到其中的一个极值。这是因为算法的每次迭代都会缩小区间的长度，并且保证区间内始终包含至少一个极值点。因此如果求得的极值点也能满足需求，则这种方法还是适用的。

对于二维甚至更高维的函数，本节的方法就不再适用了。后面会讨论其他一些方法以供参考。

6.2 无导数优化方法

无导数优化方法针对的是黑盒函数（Black-Box Function）。也就是说这类方法完全不关心函数的具体形式和特点。黑盒函数在实际问题中比较常见。这是因为目标函数往往太过复杂或者根本无法写出表达式。我们来看这样两个例子。

- 开采、排污等都会对环境造成危害，因此不能过度，但如何才算过度是很难计算的。一种方法就是提出若干种实施计划，然后通过计算机模拟，预测未来几年的经济效益与环境变化，检验是否达到经济发展与环境保护的最佳平衡点。因此目标函数过于复杂，计算函数值就相当于要做一系列复杂的计算机模拟。
- 人脸识别、文字识别等机器学习问题都是通过观察得到的特征来对结果进行推测的，而事实上观察到的特征与真实结果时间并没有一个确定性的关系。我们无法写出一个绝对正确的从特征到结果之间的关系式，也就无法写出目标函数。虽然目标函数写不出来，但目的是非常明确的，也就是推测出的结果应当同真实结果尽量一致。

本节的几种方法主要用于处理这两种情况。其中模式搜索法和坐标下降法主要针对第1种情况，代理模型法则同时适用于这两种情况。这里按照多数论文的习惯，我们考虑求最小值，即

$$\min f(x) \tag{6.1}$$

由于我们主要采用迭代法，所以一般记 x_k 是第 k 个迭代点，而迭代初始点就是 x_1。

6.2.1 模式搜索法

模式搜索（Pattern Search）是一种很简单的搜索算法。既然不知道搜索方向，那么我们就事先随机选择几个方向，或者凭经验选好几个可能的方向。之后我们尝试朝这几个方向迭代一步，如果有改进，则更新迭代点，如果没有改进，则将迭代步变短，也就是去搜索更局部的点。

算法 40 模式搜索法

输入: 初始点 x_1，搜索方向集合 $\mathcal{D}$，常数 $\epsilon > 0$

$k \leftarrow 1$，$\alpha \leftarrow 1$。
while $\alpha > \epsilon$ **do**
  $d^* \leftarrow \arg\min_{d \in \mathcal{D}} f(x_k + \alpha d)$。
  if $f(x_k + \alpha d^*) < f(x_k)$ **then**
    $x_{k+1} \leftarrow x_k + \alpha d^*$。
    $k \leftarrow k + 1$。
  else
    $\alpha \leftarrow \dfrac{\alpha}{2}$。
  end if
end while

模式搜索的实现非常简单，如果没有更好的思路，则可以尝试这样一种方法。模式搜索也可以与模拟退火、遗传算法等所谓的智能算法结合。

6.2.2 坐标下降法

坐标下降法（Coordinate Descent Method），也可以称为交替方向法（Alternating Direction Method），同模式搜索有些类似，但应用更为广泛。在每一次迭代的时候，我们只允许一个维度改变，其余维度均固定不动。这里我们记 e_i 为第 i 个分量为 1、其余分量均为 0 的列向量，也就是第 i 个坐标轴方向。我们每次迭代就是沿 e_i 进行搜索。

算法 41 坐标下降法

输入: 初始点 x_1，常数 $\epsilon > 0$。

$k \leftarrow 1$。
repeat
  $x_k^{(1)} \leftarrow x_k$。
  for $i \in \{1, 2, \ldots, n\}$ **do**
    $\alpha_k^{(i)} \leftarrow \arg\min_{\alpha} f(x_k^{(i)} + \alpha e_i)$
    $x_k^{(i+1)} \leftarrow x_k^{(i)} + \alpha_k^{(i)} e_i$。
  end for

8: $x_{k+1} \leftarrow x_k^{(n+1)}$。

9: $k \leftarrow k+1$。

10: **until** $\|x_k - x_{k-1}\| \leqslant \epsilon$

其中的第5行求迭代步长，对于黑盒函数来说可能是无法精确做到的。这时可以考虑尝试几个步长并选择最优的一个。因为这里只改变了一个维度，所以也有可能利用到一些特殊性质。比如目标函数的相反数在这个维度上是单峰函数，这时就可以运用我们之前讨论过的算法。另外一个可能性就是，虽然目标函数原本非常复杂，只能作为黑盒函数来处理，但如果仅考虑一个维度，也许目标函数就变得非常简单。这时我们就有可能很容易地找到最优迭代步长。坐标下降法的这个思想也可以进一步推广，我们可以把某几个关系密切的维度放到一起考虑，只要这时目标函数仍然比较简单，求解速度就可能会更快。

6.2.3 代理模型法

对于目标函数过于复杂或者不存在表达式的问题，我们都可以通过代理模型（Surrogate Model）来解决。代理模型是指对真实函数的一个近似。虽然真实目标函数很复杂，但我们可以想办法找到一个相对简单的代理模型去近似。如果能够找到这样一个近似，我们就可以通过找代理模型的最优解，来得到真实目标函数的近似最优解。因此代理模型法的两个关键点就是：找到好的代理模型，找到代理模型的最优解。这两个关键点也就是我们通常说的模型和算法。关于模型和算法谁更重要，也有很多争论。

在某些物理问题中，算法比模型重要。这是因为对于许多真实问题，我们已经有具体的物理公式去描述了，只是描述方式有可能是基于一些理想情况。因此可以认为真实目标函数是存在的，只是其表达式有可能过于复杂，计算目标函数有时意味着求解一个或多个偏微分方程组。我们要做的是尽可能找到真实目标函数的最优解。这时通常并不需要花费心思来找到好的代理模型，而是需要一个好的算法，在找到最优解或者近似解的过程中，尽量减少计算真实目标函数的次数，因为每计算一次真实目标函数都意味着巨大的计算量。

在某些机器学习问题中，模型比算法重要。构造代理模型常常需要考虑如何更好地反映真实情况、更多地利用先验知识等。好的模型通常在背后有一个好的故事，能够解释这样构造模型的合理性。相反，由于真实的目标函数是不存在的，所以只能去求代理模型的最优解。而使用好的算法求得代理模型的全局最优解通常也并不意味着“最优”。使用一些并不高效或者并不收敛的算法去求解，也能经常得到很不错的解。如果算法本身便于直观理解，则更容易受到工程师的欢迎。

什么是好的代理模型其实也很难判定。一类代理模型是具有普适性的，对几乎任何问题都可以尝试。另一类代理模型是具有特定性的，它往往隐含了许多先验知识。我们列举一些常见的代理模型。

- 线性模型。目标函数 $f(\boldsymbol{x})$ 与变量 $\boldsymbol{x}$ 是线性关系，即 $f(\boldsymbol{x}) = \boldsymbol{a}^{\mathrm{T}}\boldsymbol{x} + \boldsymbol{b}$。
- 多项式模型。其中最常用的是二次模型，即 $f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}^{\mathrm{T}}\boldsymbol{x} + \boldsymbol{c}$。
- 径向基函数模型。即 $f(\boldsymbol{x}) = \sum_{i=1}^{m} \lambda_i f_i(\|\boldsymbol{x} - c_i\|)$，所谓的径向基函数就是 $f_i(\|\boldsymbol{x} - c_i\|)$ 仅与变量 $\boldsymbol{x}$ 到中心点 c_i 的距离有关。

除此之外，还有神经网络模型等更为复杂的模型。这里就不一一介绍了。

大多数代理模型都有明确的表达式，形式相对简单。这时我们一般使用导数优化方法来求解。具体如何使用导数优化方法及选用哪种导数优化方法，都与模型的特点密切相关。在6.3节中我们将简要介绍一些常用的导数优化方法以供参考。

6.3 导数优化方法

如果我们面对的是一个相对简单的函数，而不再是黑盒函数，那么这时要搜索最小值，我们可以利用的信息就更多。最直接的信息是梯度，梯度是一个函数局部最快的上升方向，相反，负梯度就是一个函数局部最快的下降方向。但除了沿着负梯度方向外，也还有很多其他选择。在本节中我们主要考虑这样一个一般性的问题：

$$\min_{x} \quad f(x) \tag{6.2}$$

其中，$f(x)$ 的梯度是 $\nabla f(x)$。

导数优化方法一般都是迭代法，即从一个初始选定的 x_1 开始进行改进。在第 k 次迭代时，我们首先根据某种方法选定一个迭代方向 d_k，再确定一个实数步长 α_k，然后使用

$$x_{k+1} \leftarrow x_k + \alpha_k d_k \tag{6.3}$$

得到新的迭代点。这里式(6.3)一般被称为迭代格式。具体如何选择迭代方向 d_k 及步长 α_k 都是本节将要讨论的问题。

6.3.1 线搜索

我们首先研究在迭代法中如何选择步长。

在理想情况下，我们选择一个步长使得目标函数值达到最小，即 $\alpha_k = \arg\min_\alpha f(x_k + \alpha d_k)$。如果 $f(x)$ 的表达式比较简单，则这样确实是一个很不错的选择。但如果 $f(x)$ 的表达式稍微复杂一些，则求得 α_k 的难度就太大了。这样选取的 α_k 可以使 $f(x_k + \alpha d_k)$ 关于 α 的偏导数为 0，即 $d_k^T \nabla f(x_k + \alpha d_k) = 0$ 或者写为 $d_k^T \nabla f(x_{k+1}) = 0$。也就是说，当前点 x_k 处的迭代方向 d_k 和下一点 x_{k+1} 处的梯度方向 $\nabla f(x_{k+1})$ 是垂直的。这种线搜索被称为精确线搜索，因为它使得目标函数在搜索方向上精确地达到最小。相反，其他线搜索都属于非精确线搜索。

最简单的步长选取方式是固定步长，例如选定 $\alpha_k = 1$。另一种很简单的方式是以固定速率逐渐缩小步长，例如 $\alpha_k = 0.001 \times 0.995^k$，这种步长选取方式在工业界很常见。这种步长一般来说不会有理论上的收敛性质，需要工程师根据经验选取一个好用的步长公式。

还有一类方法，就是通过搜索得到满足某个条件的步长。前面给出的步长都不能保证 x_{k+1} 处的目标函数值比 x_k 处的目标函数值有所改进。实际上这个要求基本上只有通过搜索步长才能满足。常用的步长条件有如下三种。

$$f(x_k + \alpha_k d_k) \leqslant f(x_k) + c_1 \alpha_k d_k \nabla f(x_k) \tag{6.4}$$

$$d_k^T \nabla f(x_k + \alpha_k d_k) \geqslant c_2 d_k^T \nabla f(x_k) \tag{6.5}$$

$$|d_k^T \nabla f(x_k + \alpha_k d_k)| \leqslant c_2 |d_k^T \nabla f(x_k)| \tag{6.6}$$

式(6.4)成立时我们称之为满足 Armijo 条件。式(6.4)和式(6.5)同时成立时我们称之为满足 Wolfe 条件。式 (6.4) 和式 (6.6) 同时成立时我们称之为满足强 Wolfe 条件。我们可以看到，满足式 (6.6) 就一定满足式 (6.5)，所以满足强 Wolfe 条件就一定满足 Wolfe 条件，而满足 Wolfe 条件也一定满足 Armijo 条件。

其中最简单也最常用的就是 Armijo 线搜索，Armijo 线搜索的算法流程如下。

算法 42 Armijo 线搜索

1: $\alpha_k \leftarrow 1$。
2: **while** α_k 不满足式 (6.4) **do**
3: $\quad \alpha_k \leftarrow \alpha_k / 2$。
4: **end while**

Wolfe 线搜索和强 Wolfe 线搜索在理论分析中非常重要，但实际计算效率并不一定会有所提高。

6.3.2 梯度下降法

梯度下降法（Gradient Descent Method）也叫作最速下降法（Steepest Descent Method），这是因为负梯度是一个函数局部最快的下降方向。

梯度下降法的迭代格式为

$$x_{k+1} \leftarrow x_k - \alpha_k \nabla f(x_k) \tag{6.7}$$

梯度下降法非常简单，只需要知道如何计算目标函数的梯度就可以写出迭代格式。因此，尽管在不少情况下梯度下降法的收敛速度都很慢，也依然不影响它在工业界的广泛应用。梯度下降法应用到一些具体模型上有时也会被视作一类特定的算法，例如神经网络中的后向传导算法（Back Propagation Algorithm）。

在机器学习中经常有 $f(x) = \sum_{i=1}^{m} l_i(x)$，其中 $l_i(x)$ 是第 i 个训练样本的损失函数。这时我们可以使用随机梯度下降法（Stochastic Gradient Descent Method）。其迭代格式为

$$x_{k+1} \leftarrow x_k - \alpha_k \nabla l_r(x_k) \tag{6.8}$$

其中 $r \in \{1, 2, \cdots, m\}$ 为随机数。这种做法可以理解为随机选择一个训练样本，进行一次梯度下降法的迭代。在机器学习的问题中，我们通常不需要真的求得最优值。这样不精确的迭代，使得算法不容易过拟合。由于随机梯度下降法的普及，与此相对的普通梯度下降法有时被称为批量梯度下降法（Batch Gradient Descent Method），因为它同时考虑所有训练样本。介于批量梯度下降法和随机梯度下降法之间，还有小批量梯度下降法（Mini-Batch Gradient Descent Method），也就是每次迭代选择若干个训练样本。

除了之前介绍的一般性的步长选取方法，对于梯度法我们再介绍一种特殊的步长计算公式。该方法是由 [BB88] 提出的，现在一般被称为 Barzilai-Borwein 步长或 BB 步长。BB 步长有两个公式，选择其一即可。

$$\alpha_k = \frac{(\nabla f(x_k) - \nabla f(x_{k-1}))^T (x_k - x_{k-1})}{(\nabla f(x_k) - \nabla f(x_{k-1}))^T (\nabla f(x_k) - \nabla f(x_{k-1}))}$$
$$\alpha_k = \frac{(x_k - x_{k-1})^T (x_k - x_{k-1})}{(x_k - x_{k-1})^T (\nabla f(x_k) - \nabla f(x_{k-1}))} \tag{6.9}$$

BB 步长适合我们在对步长选择缺乏经验的时候尝试，这经常会有不错的效果。在后面我们还会提到 BB 步长，大致解释这两个公式是如何得到的。

6.3.3 共轭梯度法

共轭梯度法（Conjugate Gradient Method）有许多种推导方式，有的书中从 Krylov 子空间出发进行推导，有的书中从共轭的定义出发进行推导。不同的推导方式源于不同的理解。这里我们选择一个相对简单的出发点，便于读者自行推导共轭梯度法的迭代格式。

共轭梯度法最早用于求解正定线性方程组，或者说求解严格凸二次函数的最小值，后来推广到一般目标函数。这里我们也先考虑求解严格凸二次函数的最小值。记我们要求解的目标函数是

$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}^{\mathrm{T}}\boldsymbol{x} \tag{6.10}$$

其中矩阵 $\boldsymbol{A}$ 是正定矩阵。很容易知道，求解这一最优化问题等价于求解 $\nabla f(\boldsymbol{x}) = 0$，即 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$。因此也可以认为是求解正定线性方程组。我们假设在迭代算法中始终采用精确线搜索，那么对于固定的 $\boldsymbol{x}_k$ 和 $\boldsymbol{d}_k$，步长 α_k 应当使得 $f(\boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k)$ 达到最小。

$$\begin{aligned} & f(\boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k) \\ =& \frac{1}{2}(\boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k)^{\mathrm{T}} \boldsymbol{A}(\boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k) - \boldsymbol{b}^{\mathrm{T}}(\boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k) \\ =& \frac{1}{2}\boldsymbol{d}_k^{\mathrm{T}}\boldsymbol{A}\boldsymbol{d}_k \alpha_k^2 + (\boldsymbol{A}\boldsymbol{x}_k - \boldsymbol{b})^{\mathrm{T}}\boldsymbol{d}_k \alpha_k + \frac{1}{2}\boldsymbol{x}_k^{\mathrm{T}}\boldsymbol{A}\boldsymbol{x}_k - \boldsymbol{b}^{\mathrm{T}}\boldsymbol{x}_k \end{aligned} \tag{6.11}$$

这是一个关于 α_k 的一元二次函数，二次项系数 $\frac{1}{2}\boldsymbol{d}_k^{\mathrm{T}}\boldsymbol{A}\boldsymbol{d}_k > 0$，因此在对称轴处达到最小。所以我们有

$$\alpha_k = -\frac{(\boldsymbol{A}\boldsymbol{x}_k - \boldsymbol{b})^{\mathrm{T}}\boldsymbol{d}_k}{\boldsymbol{d}_k^{\mathrm{T}}\boldsymbol{A}\boldsymbol{d}_k} \tag{6.12}$$

如果我们使用的是梯度下降法，则应当选取 $\boldsymbol{d}_k = -\nabla f(\boldsymbol{x}_k) = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_k$。在共轭梯度法中，我们考虑对 $\boldsymbol{d}_k$ 进行一个修正，使得下降速度更快。一种简单的修正方式就是借助上

一次的迭代方向 $\boldsymbol{d}_{k-1}$。也就是说，我们令 $\boldsymbol{d}_k = -\nabla f(\boldsymbol{x}_k) + \beta_k \boldsymbol{d}_{k-1}$。与求 α_k 采用同样的思路，我们也可以求得 β_k。

$$
\begin{aligned}
& f(\boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k) \\
=& f(\boldsymbol{x}_k + \alpha_k(-\nabla f(\boldsymbol{x}_k) + \beta_k \boldsymbol{d}_{k-1})) \\
=& \frac{1}{2}(\boldsymbol{x}_k + \alpha_k(-\nabla f(\boldsymbol{x}_k) + \beta_k \boldsymbol{d}_{k-1}))^{\mathrm{T}} \boldsymbol{A}(\boldsymbol{x}_k + \alpha_k(-\nabla f(\boldsymbol{x}_k) + \beta_k \boldsymbol{d}_{k-1})) \\
& - \boldsymbol{b}^{\mathrm{T}}(\boldsymbol{x}_k + \alpha_k(-\nabla f(\boldsymbol{x}_k) + \beta_k \boldsymbol{d}_{k-1})) \\
=& \frac{1}{2}\boldsymbol{d}_{k-1}^{\mathrm{T}} \boldsymbol{A} \boldsymbol{d}_{k-1} \alpha_k^2 \beta_k^2 + \alpha_k(\boldsymbol{A}(\boldsymbol{x}_k - \alpha_k \nabla f(\boldsymbol{x}_k)) - \boldsymbol{b})^{\mathrm{T}} \boldsymbol{d}_{k-1} \beta_k \\
& + \frac{1}{2}(\boldsymbol{x}_k - \alpha_k \nabla f(\boldsymbol{x}_k))^{\mathrm{T}} \boldsymbol{A}(\boldsymbol{x}_k - \alpha_k \nabla f(\boldsymbol{x}_k)) - \boldsymbol{b}^{\mathrm{T}}(\boldsymbol{x}_k - \alpha_k \nabla f(\boldsymbol{x}_k))
\end{aligned}
$$

这是一个关于 β_k 的一元二次函数，二次项系数 $\frac{1}{2}\boldsymbol{d}_{k-1}^{\mathrm{T}} \boldsymbol{A} \boldsymbol{d}_{k-1} \alpha_k^2 > 0$，因此在对称轴处达到最小。所以我们有

$$
\beta_k = \frac{(\boldsymbol{A}(\boldsymbol{x}_k - \alpha_k \nabla f(\boldsymbol{x}_k)) - \boldsymbol{b})^{\mathrm{T}} \boldsymbol{d}_{k-1}}{\boldsymbol{d}_{k-1}^{\mathrm{T}} \boldsymbol{A} \boldsymbol{d}_{k-1} \alpha_k} \tag{6.13}
$$

后面我们还会对 β_k 的这个表达式进一步化简。

考虑到 $\boldsymbol{d}_{k-1}$ 是上一次的迭代方向，如果现在还在朝这个方向迭代，则应当不会有任何改进。也就是说 $f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_{k-1})$ 应当在 $\alpha = 0$ 时取得最小值。而

$$
\begin{aligned}
& f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_{k-1}) \\
=& \frac{1}{2}(\boldsymbol{x}_k + \alpha \boldsymbol{d}_{k-1})^{\mathrm{T}} \boldsymbol{A}(\boldsymbol{x}_k + \alpha \boldsymbol{d}_{k-1}) - \boldsymbol{b}^{\mathrm{T}}(\boldsymbol{x}_k + \alpha \boldsymbol{d}_{k-1}) \\
=& \frac{1}{2}\boldsymbol{d}_{k-1}^{\mathrm{T}} \boldsymbol{A} \boldsymbol{d}_{k-1} \alpha^2 + (\boldsymbol{A}\boldsymbol{x}_k - \boldsymbol{b})^{\mathrm{T}} \boldsymbol{d}_{k-1} \alpha + \frac{1}{2}\boldsymbol{x}_k^{\mathrm{T}} \boldsymbol{A} \boldsymbol{x}_k - \boldsymbol{b}^{\mathrm{T}} \boldsymbol{x}_k
\end{aligned} \tag{6.14}
$$

是关于 α 的二次函数，在 $\alpha = \dfrac{(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_k)^{\mathrm{T}} \boldsymbol{d}_{k-1}}{\boldsymbol{d}_{k-1}^{\mathrm{T}} \boldsymbol{A} \boldsymbol{d}_{k-1}}$ 时取得最小值。我们知道 $\alpha = 0$ 时应当取得最小值，因此

$$
(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_k)^{\mathrm{T}} \boldsymbol{d}_{k-1} = 0 \tag{6.15}
$$

将其代回到 β_k 的表达式，可以进一步化简得到

$$
\beta_k = \frac{\boldsymbol{d}_{k-1}^{\mathrm{T}} \boldsymbol{A} \nabla f(\boldsymbol{x}_k)}{\boldsymbol{d}_{k-1}^{\mathrm{T}} \boldsymbol{A} \boldsymbol{d}_{k-1}} \tag{6.16}
$$

为什么叫共轭梯度法呢？这主要是因为

$$\begin{aligned}\boldsymbol{d}_{k-1}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{d}_k &= \boldsymbol{d}_{k-1}^{\mathrm{T}}\boldsymbol{A}(-\nabla f(\boldsymbol{x}_k)+\beta_k\boldsymbol{d}_{k-1})\\ &= -\boldsymbol{d}_{k-1}^{\mathrm{T}}\boldsymbol{A}\nabla f(\boldsymbol{x}_k)+\beta_k\boldsymbol{d}_{k-1}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{d}_{k-1}\\ &= 0\end{aligned} \tag{6.17}$$

直接代入 β_k 的表达式即可验证最后一步成立。前后两个迭代方向满足 $\boldsymbol{d}_{k-1}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{d}_k=0$，这时我们就称 $\boldsymbol{d}_{k-1}$ 和 $\boldsymbol{d}_k$ 关于 $\boldsymbol{A}$ 是共轭的。

如果目标函数不是严格凸二次函数，那么前面所给出的 β_k 的表达式就不再适用。但是我们可以等价地改写其表达式，使其能够推广到一般目标函数。具体思路是设法通过等价变换，使得在 β_k 的表达式中仅包含 $\nabla f(\boldsymbol{x}_k)$、$\boldsymbol{d}_{k-1}$ 等，这样 β_k 就不拘泥于 $f(\boldsymbol{x}_k)$ 的具体形式了。对比 α_{k-1} 和 β_k 的表达式，它们的分母相同，因此我们可以得到二者之间的关系：

$$-\beta_k(\boldsymbol{A}\boldsymbol{x}_{k-1}-\boldsymbol{b})^{\mathrm{T}}\boldsymbol{d}_{k-1}=\alpha_{k-1}\boldsymbol{d}_{k-1}^{\mathrm{T}}\boldsymbol{A}\nabla f(\boldsymbol{x}_k) \tag{6.18}$$

我们知道 $\boldsymbol{A}\boldsymbol{x}_{k-1}-\boldsymbol{b}=\nabla f(\boldsymbol{x}_{k-1})$，所以有

$$\begin{aligned}-\beta_k(\nabla f(\boldsymbol{x}_{k-1}))^{\mathrm{T}}\boldsymbol{d}_{k-1} &= -\beta_k(\boldsymbol{A}\boldsymbol{x}_{k-1}-\boldsymbol{b})^{\mathrm{T}}\boldsymbol{d}_{k-1}\\ &= \alpha_{k-1}\boldsymbol{d}_{k-1}^{\mathrm{T}}\boldsymbol{A}\nabla f(\boldsymbol{x}_k)\\ &= \alpha_{k-1}(\nabla f(\boldsymbol{x}_k))^{\mathrm{T}}\boldsymbol{A}\boldsymbol{d}_{k-1}\\ &= (\nabla f(\boldsymbol{x}_k))^{\mathrm{T}}\boldsymbol{A}(\alpha_{k-1}\boldsymbol{d}_{k-1})\\ &= (\nabla f(\boldsymbol{x}_k))^{\mathrm{T}}\boldsymbol{A}(\boldsymbol{x}_k-\boldsymbol{x}_{k-1})\\ &= (\nabla f(\boldsymbol{x}_k))^{\mathrm{T}}(\nabla f(\boldsymbol{x}_k)-\nabla f(\boldsymbol{x}_{k-1}))\end{aligned} \tag{6.19}$$

从而

$$\beta_k=\frac{(\nabla f(\boldsymbol{x}_k))^{\mathrm{T}}(\nabla f(\boldsymbol{x}_k)-\nabla f(\boldsymbol{x}_{k-1}))}{(\nabla f(\boldsymbol{x}_{k-1}))^{\mathrm{T}}\boldsymbol{d}_{k-1}} \tag{6.20}$$

得到了一个和 $f(\boldsymbol{x}_k)$ 的具体形式无关的 β_k 的表达式，因此可以直接推广到一般函数。式(6.20)也被称为共轭下降公式[Fle13]。

常用的 β_k 的表达式不止式(6.20)一种，我们还有

- Fletcher-Reeves 公式[FR64]：$\beta_k=\dfrac{\|\nabla f(\boldsymbol{x}_k)\|_2^2}{\|\nabla f(\boldsymbol{x}_{k-1})\|_2^2}$。
- Polak-Ribière-Polyak 公式[Pol69]：$\beta_k=\dfrac{(\nabla f(\boldsymbol{x}_k)-\nabla f(\boldsymbol{x}_{k-1}))^{\mathrm{T}}\nabla f(\boldsymbol{x}_k)}{\|\nabla f(\boldsymbol{x}_{k-1})\|_2^2}$。

- Hestenes-Stiefel 公式[HS52]：$\beta_k = \dfrac{(\nabla f(\boldsymbol{x}_k) - \nabla f(\boldsymbol{x}_{k-1}))^{\mathrm{T}} \nabla f(\boldsymbol{x}_k)}{\boldsymbol{d}_{k-1}^{\mathrm{T}}(\nabla f(\boldsymbol{x}_k) - \nabla f(\boldsymbol{x}_{k-1}))}$。
- Dai-Yuan 公式[DY99]：$\beta_k = \dfrac{\|\nabla f(\boldsymbol{x}_k)\|_2^2}{\boldsymbol{d}_{k-1}^{\mathrm{T}}(\nabla f(\boldsymbol{x}_k) - \nabla f(\boldsymbol{x}_{k-1}))}$。

有意思的是，当 $f(\boldsymbol{x})$ 是凸二次目标函数时，无论选用哪一个 β_k 的表达式都是等价的。

我们将共轭梯度法的算法流程整理如下。

算法 43 共轭梯度法

输入： $x_1 \in \mathcal{R}^n$，$\epsilon > 0$。

$k \leftarrow 1$。
while $\|\nabla f(\boldsymbol{x}_k)\|_2 > \epsilon$ **do**
  if $k = 1$ **then**
    $d_k = -\boldsymbol{g}_k$。
  else
    使用式 (6.20) 或其他公式计算 β_k。
    $\boldsymbol{d}_k = -\boldsymbol{g}_k + \beta_k \boldsymbol{d}_{k-1}$。
  end if
  使用某种线搜索方法计算步长 α_k。
  $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k$。
  $k \leftarrow k + 1$。
end while

当 $f(\boldsymbol{x})$ 是严格凸二次函数时，共轭梯度法还有很多有趣的性质，我们简单列举如下。

- 任意两个迭代方向都是共轭的，即 $\boldsymbol{d}_i^{\mathrm{T}} \boldsymbol{A} \boldsymbol{d}_j = 0, \forall i \neq j$。
- 任意两个梯度方向都是正交的，即 $(\nabla f(\boldsymbol{x}_i))^{\mathrm{T}} \nabla f(\boldsymbol{x}_j) = 0, \forall i \neq j$。
- 迭代至多 n 次，就会得到最优解。然而梯度下降法并没有这个性质。

虽然 $f(\boldsymbol{x})$ 是一般函数时不再具有这些性质，但共轭梯度法常常有不错的表现。

6.3.4 牛顿法

牛顿方向和梯度方向最大的差别是考虑了 Hessian 矩阵。我们记 $\boldsymbol{x}_k$ 处的 Hessian 矩阵为 $\nabla^2 f(\boldsymbol{x}_k)$。

我们求 $f(\boldsymbol{x})$ 的最值点比较困难，因此考虑求稳定点，即 $\nabla f(\boldsymbol{x}) = 0$ 的解。如果记 $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{d}_k$，则根据泰勒展开，有 $\nabla f(\boldsymbol{x}_{k+1}) \approx \nabla f(\boldsymbol{x}_k) + \nabla^2 f(\boldsymbol{x}_k)\boldsymbol{d}_k$。我们希望 $\nabla f(\boldsymbol{x}_{k+1}) = 0$，也就可以近似地认为 $\nabla f(\boldsymbol{x}_k) + \nabla^2 f(\boldsymbol{x}_k)\boldsymbol{d}_k = 0$。如果 $\nabla^2 f(\boldsymbol{x}_k)$ 可逆，则我们可以得到 $\boldsymbol{d}_k = -(\nabla^2 f(\boldsymbol{x}_k))^{-1}\nabla f(\boldsymbol{x}_k)$。因此牛顿法的搜索方向是 $-(\nabla^2 f(\boldsymbol{x}_k))^{-1} f(\boldsymbol{x}_k)$，其迭代格式为

$$\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k - \alpha_k(\nabla^2 f(\boldsymbol{x}_k))^{-1}\nabla f(\boldsymbol{x}_k) \tag{6.21}$$

这里的步长 α_k 也有多种取法。但与梯度下降法不同的是，这里步长取 1 的效果通常不错。值得注意的是，虽然我们写作 $\boldsymbol{d}_k = -(\nabla^2 f(\boldsymbol{x}_k))^{-1}\nabla f(\boldsymbol{x}_k)$，但在计算 $\boldsymbol{d}_k$ 时，并不真正求逆，而是去求解线性方程组 $(\nabla^2 f(\boldsymbol{x}_k))\boldsymbol{d}_k = -\nabla f(\boldsymbol{x}_k)$。

在解方程的过程中也有一个牛顿法，实际上和这里说的牛顿法是一回事。我们这里的目标虽然是求解 $f(\boldsymbol{x})$ 的最小值，但事实上我们的方法是用于求解 $\nabla f(\boldsymbol{x}) = 0$ 的，也可以认为是在求解方程组。假设我们要求解一元方程 $f(x) = 0$，则我们可以将式(6.21)中的 $\nabla f(\boldsymbol{x})$ 替换为 $f(x)$，将 $\nabla^2 f(\boldsymbol{x})$ 替换为 $f'(x)$，牛顿法的迭代格式变为

$$x_{k+1} \leftarrow x_k - \alpha_k \frac{f(x_k)}{f'(x_k)} \tag{6.22}$$

其中 $f'(x_k)$ 是 x_k 处的导数。

牛顿法在计算上有以下局限性。

- 计算矩阵 $\nabla^2 f(\boldsymbol{x}_k)$ 可能要花费很长时间。
- 可能没有足够的内存去存储矩阵 $\nabla^2 f(\boldsymbol{x}_k)$。
- $\nabla^2 f(\boldsymbol{x}_k)$ 不一定可逆，$(\nabla^2 f(\boldsymbol{x}_k))^{-1}$ 也就不一定存在。

因此一般只有当问题规模较小，而且 $f(\boldsymbol{x})$ 是严格凸函数的时候，我们才会考虑牛顿法。在其他情形下使用牛顿法时，都需要设法进行一些修正。

6.3.5 拟牛顿法

牛顿法的局限性基本上源于 $\nabla^2 f(\boldsymbol{x}_k)$。在拟牛顿法中，我们不再直接使用 $\nabla^2 f(\boldsymbol{x}_k)$，而是采用其他方式替代。

下面我们用 $\boldsymbol{B}_k$ 近似取代 $\nabla^2 f(\boldsymbol{x}_k)$，用 $\boldsymbol{H}_k$ 近似取代 $(\nabla^2 f(\boldsymbol{x}_k))^{-1}$，然后研究如何选取 $\boldsymbol{B}_k$ 和 $\boldsymbol{H}_k$。在第 1 次迭代的时候，我们没有任何有效信息可以用于选取 $\boldsymbol{B}_0$ 和 $\boldsymbol{H}_0$，因此一

般直接取 $\boldsymbol{B}_0 = \boldsymbol{H}_0 = \boldsymbol{I}$，即单位矩阵。在之后的每一次迭代中，我们可以根据当前信息设法找到合适的 $\boldsymbol{B}_k$ 和 $\boldsymbol{H}_k$。

根据泰勒展开，我们知道 $\nabla f(\boldsymbol{x}_{k+1}) \approx \nabla f(\boldsymbol{x}_k) + \nabla^2 f(\boldsymbol{x}_k)(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k)$，所以我们可以假设

$$\nabla f(\boldsymbol{x}_{k+1}) = \nabla f(\boldsymbol{x}_k) + \boldsymbol{B}_{k+1}(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k) \tag{6.23}$$

$$\nabla f(\boldsymbol{x}_{k+1}) = \nabla f(\boldsymbol{x}_k) + \boldsymbol{H}_{k+1}^{-1}(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k) \tag{6.24}$$

式(6.23)和式(6.24)可以进一步写为

$$\boldsymbol{B}_{k+1}(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k) = \nabla f(\boldsymbol{x}_{k+1}) - \nabla f(\boldsymbol{x}_k) \tag{6.25}$$

$$\boldsymbol{H}_{k+1}(\nabla f(\boldsymbol{x}_{k+1}) - \nabla f(\boldsymbol{x}_k)) = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k \tag{6.26}$$

式(6.25)和式(6.26)被称为拟牛顿公式（Quasi-Newton Formula）。凡是满足拟牛顿公式的方法都可以被称为拟牛顿方法。为了后面书写方便，我们定义 $\boldsymbol{s}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$，$\boldsymbol{y}_k = \nabla f(\boldsymbol{x}_{k+1}) - \nabla f(\boldsymbol{x}_k)$。这样，两个拟牛顿公式可以简化为

$$\boldsymbol{B}_{k+1}\boldsymbol{s}_k = \boldsymbol{y}_k \tag{6.27}$$

$$\boldsymbol{H}_{k+1}\boldsymbol{y}_k = \boldsymbol{s}_k \tag{6.28}$$

满足拟牛顿公式的矩阵非常多，我们先来考虑其中尽可能简单的矩阵。这里我们直接给出两个，很容易验证它们满足第 1 个拟牛顿公式。

$$\frac{\boldsymbol{y}_k \boldsymbol{s}_k^{\mathrm{T}}}{\boldsymbol{s}_k^{\mathrm{T}} \boldsymbol{s}_k} \tag{6.29}$$

$$\frac{\boldsymbol{y}_k \boldsymbol{y}_k^{\mathrm{T}}}{\boldsymbol{y}_k^{\mathrm{T}} \boldsymbol{s}_k} \tag{6.30}$$

我们一般选择式 (6.30)，这是因为它是对称矩阵。同样，我们可以给出两个类似的矩阵来满足第 2 个拟牛顿公式。

$$\frac{\boldsymbol{s}_k \boldsymbol{y}_k^{\mathrm{T}}}{\boldsymbol{y}_k^{\mathrm{T}} \boldsymbol{y}_k} \tag{6.31}$$

$$\frac{\boldsymbol{s}_k \boldsymbol{s}_k^{\mathrm{T}}}{\boldsymbol{s}_k^{\mathrm{T}} \boldsymbol{y}_k} \tag{6.32}$$

注意到这两个矩阵和之前介绍过的 Barzilai-Borwein 步长公式非常相似。可以验证，$\frac{\boldsymbol{s}_k\boldsymbol{y}_k^{\mathrm{T}}}{\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{y}_k}$ 的唯一非零特征值是 $\frac{\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k}{\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{y}_k}$，$\frac{\boldsymbol{s}_k\boldsymbol{s}_k^{\mathrm{T}}}{\boldsymbol{s}_k^{\mathrm{T}}\boldsymbol{y}_k}$ 的唯一非零特征值是 $\frac{\boldsymbol{s}_k^{\mathrm{T}}\boldsymbol{s}_k}{\boldsymbol{s}_k^{\mathrm{T}}\boldsymbol{y}_k}$。对比式 (6.9) 可以发现，这两个非零特征值就是 BB 步长。事实上，BB 步长就是这样构造的。

上面给出的矩阵虽然满足拟牛顿公式，却与 $\boldsymbol{B}_k$ 和 $\boldsymbol{H}_k$ 毫无关系。假设在前面的迭代中，我们已经找到了比较准确的 $\boldsymbol{B}_k$ 和 $\boldsymbol{H}_k$，那么将它们微调成为 $\boldsymbol{B}_{k+1}$、$\boldsymbol{H}_{k+1}$ 是比较好的策略。这是因为两次迭代点一般比较近，它们的 Hessian 矩阵差别不会太大。

我们先来考虑修正 $\boldsymbol{B}_k$ 为 $\boldsymbol{B}_{k+1}$，修正方式是不唯一的，我们这里只考虑一种思路。刚才我们已经指出 $\boldsymbol{B}_{k+1}$ 可以选为 $\frac{\boldsymbol{y}_k\boldsymbol{y}_k^{\mathrm{T}}}{\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k}$，而我们又希望它接近 $\boldsymbol{B}_k$，所以我们从它们二者出发进行修正。令 $\boldsymbol{B}_{k+1}=\boldsymbol{B}_k+\frac{\boldsymbol{y}_k\boldsymbol{y}_k^{\mathrm{T}}}{\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k}-\boldsymbol{M}$，我们只需要选择一个 $\boldsymbol{M}$，让 $\boldsymbol{B}_{k+1}$ 满足拟牛顿公式 (6.27)。从而我们有 $(\boldsymbol{B}_k+\frac{\boldsymbol{y}_k\boldsymbol{y}_k^{\mathrm{T}}}{\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k}-\boldsymbol{M})\boldsymbol{s}_k=\boldsymbol{y}_k$。化简后，我们得到 $\boldsymbol{M}\boldsymbol{s}_k=\boldsymbol{B}_k\boldsymbol{s}_k$。这里 $\boldsymbol{M}$ 的选择不是唯一的，如果取 $\boldsymbol{M}=\boldsymbol{B}_k$，那么又回到了之前的公式，$\boldsymbol{B}_{k+1}$ 与 $\boldsymbol{B}_k$ 无关。我们如果把 $\boldsymbol{B}_k\boldsymbol{s}_k$ 看成一个整体，则可以把 $\boldsymbol{M}$ 选成类似式 (6.30) 的形式，即

$$\boldsymbol{M}=\frac{\boldsymbol{B}_k\boldsymbol{s}_k\boldsymbol{s}_k^{\mathrm{T}}\boldsymbol{B}_k^{\mathrm{T}}}{\boldsymbol{s}_k^{\mathrm{T}}\boldsymbol{B}_k^{\mathrm{T}}\boldsymbol{s}_k} \tag{6.33}$$

于是我们有

$$\boldsymbol{B}_{k+1}=\boldsymbol{B}_k+\frac{\boldsymbol{y}_k\boldsymbol{y}_k^{\mathrm{T}}}{\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k}-\frac{\boldsymbol{B}_k\boldsymbol{s}_k\boldsymbol{s}_k^{\mathrm{T}}\boldsymbol{B}_k^{\mathrm{T}}}{\boldsymbol{s}_k^{\mathrm{T}}\boldsymbol{B}_k^{\mathrm{T}}\boldsymbol{s}_k} \tag{6.34}$$

在实际迭代中，我们需要用的是 $\boldsymbol{H}_k$ 而不是 $\boldsymbol{B}_k$。考虑 $\boldsymbol{H}_k=\boldsymbol{B}_k^{-1}$，我们可以通过 Sherman Morrison 公式求出 $\boldsymbol{B}_{k+1}^{-1}$ 作为 $\boldsymbol{H}_{k+1}$。

定理 6.2（Sherman Morrison 公式） 如果矩阵 $\boldsymbol{A}$ 可逆，$\boldsymbol{u},\boldsymbol{v}$ 都是列向量，并假设 $1+\boldsymbol{v}^{\mathrm{T}}\boldsymbol{A}^{-1}\boldsymbol{u}\neq 0$，则我们有

$$(\boldsymbol{A}+\boldsymbol{u}^{\mathrm{T}}\boldsymbol{v})^{-1}=\boldsymbol{A}^{-1}-\frac{\boldsymbol{A}^{-1}\boldsymbol{u}\boldsymbol{v}^{\mathrm{T}}\boldsymbol{A}^{-1}}{1+\boldsymbol{v}^{\mathrm{T}}\boldsymbol{A}^{-1}\boldsymbol{u}}$$

读者可以参考上述定理，对式(6.34)两边同时求逆，得到 $\boldsymbol{H}_{k+1}$ 的递推式。这里我们直接给出结果。

$$\boldsymbol{H}_{k+1}=(\boldsymbol{I}-\frac{\boldsymbol{s}_k\boldsymbol{y}_k^{\mathrm{T}}}{\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k})\boldsymbol{H}_k(\boldsymbol{I}-\frac{\boldsymbol{y}_k\boldsymbol{s}_k^{\mathrm{T}}}{\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k})+\frac{\boldsymbol{s}_k\boldsymbol{s}_k^{\mathrm{T}}}{\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k} \tag{6.35}$$

式(6.35)就是著名的 BFGS（Broyden-Fletcher Goldfarb Shanno）公式[Bro70, Fle70, Gol70, Sha70]。

类似地，我们还可以考虑直接修正 $\boldsymbol{H}_k$ 为 $\boldsymbol{H}_{k+1}$，令 $\boldsymbol{H}_{k+1} = \boldsymbol{H}_k + \frac{\boldsymbol{s}_k \boldsymbol{s}_k^{\mathrm{T}}}{\boldsymbol{s}_k^{\mathrm{T}} \boldsymbol{y}_k} - \boldsymbol{M}$。与前面推导类似，我们可以选

$$\boldsymbol{M} = \frac{\boldsymbol{H}_k \boldsymbol{y}_k \boldsymbol{y}_k^{\mathrm{T}} \boldsymbol{H}_k^{\mathrm{T}}}{\boldsymbol{y}_k^{\mathrm{T}} \boldsymbol{H}_k^{\mathrm{T}} \boldsymbol{y}_k} \tag{6.36}$$

于是我们有

$$\boldsymbol{H}_{k+1} = \boldsymbol{H}_k + \frac{\boldsymbol{s}_k \boldsymbol{s}_k^{\mathrm{T}}}{\boldsymbol{s}_k^{\mathrm{T}} \boldsymbol{y}_k} - \frac{\boldsymbol{H}_k \boldsymbol{y}_k \boldsymbol{y}_k^{\mathrm{T}} \boldsymbol{H}_k^{\mathrm{T}}}{\boldsymbol{y}_k^{\mathrm{T}} \boldsymbol{H}_k^{\mathrm{T}} \boldsymbol{y}_k} \tag{6.37}$$

式(6.37)就是著名的 DFP（Davidon Fletcher Powell）公式[Dav59, FP63]。

我们将拟牛顿法的一般流程总结如下。

算法 44 拟牛顿法一般框架

输入： $x_1 \in \mathcal{R}^n$，$\epsilon > 0$。

1: $k \leftarrow 1$，$\boldsymbol{H}_1 \leftarrow \boldsymbol{I}$。
2: **while** $\|\nabla f(\boldsymbol{x}_k)\|_2 > \epsilon$ **do**
3: 使用某种线搜索方法计算步长 α_k。
4: $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k \boldsymbol{H}_k \nabla f(\boldsymbol{x}_k)$。
5: 使用某一个公式计算 $\boldsymbol{H}_{k+1}$。
6: $k \leftarrow k + 1$。
7: **end while**

拟牛顿法很好地解决了 Hessian 矩阵的计算问题，但是仍然没有解决存储问题。尽管 $\boldsymbol{H}_1 = \boldsymbol{I}$ 是稀疏矩阵，可以节约存储，但 $\boldsymbol{H}_2$ 已经不能保证是稀疏矩阵了。我们在下面介绍的有限内存 BFGS 算法[Noc80] 在 BFGS 算法的基础上进行了修改，解决了存储问题。为了描述方便，我们先定义一些记号。

$$\begin{aligned} \rho_k &= \frac{1}{\boldsymbol{y}_k^{\mathrm{T}} \boldsymbol{s}_k} \\ \boldsymbol{V}_k &= \boldsymbol{I} - \frac{\boldsymbol{y}_k \boldsymbol{s}_k^{\mathrm{T}}}{\boldsymbol{y}_k^{\mathrm{T}} \boldsymbol{s}_k} \end{aligned} \tag{6.38}$$

于是 BFGS 公式(6.35)可以写为

$$\boldsymbol{H}_{k+1} = \boldsymbol{V}_k^{\mathrm{T}} \boldsymbol{H}_k \boldsymbol{V}_k + \boldsymbol{s}_k \rho_k \boldsymbol{s}_k^{\mathrm{T}} \tag{6.39}$$

我们可以利用式(6.39)反复代入，从 $\boldsymbol{H}_k$ 一直推至 $\boldsymbol{H}_{k-m}$。

$$\begin{aligned}
\boldsymbol{H}_k =& \boldsymbol{V}_{k-1}^{\mathrm{T}} \boldsymbol{H}_{k-1} \boldsymbol{V}_{k-1} \\
&+ \boldsymbol{s}_{k-1} \rho_{k-1} \boldsymbol{s}_{k-1}^{\mathrm{T}} \\
=& \boldsymbol{V}_{k-1}^{\mathrm{T}} \boldsymbol{V}_{k-2}^{\mathrm{T}} \boldsymbol{H}_{k-2} \boldsymbol{V}_{k-2} \boldsymbol{V}_{k-1} \\
&+ \boldsymbol{V}_{k-1}^{\mathrm{T}} \boldsymbol{s}_{k-2} \rho_{k-2} \boldsymbol{s}_{k-2}^{\mathrm{T}} \boldsymbol{V}_{k-1} \\
&+ \boldsymbol{s}_{k-1} \rho_{k-1} \boldsymbol{s}_{k-1}^{\mathrm{T}} \\
=& \boldsymbol{V}_{k-1}^{\mathrm{T}} \boldsymbol{V}_{k-2}^{\mathrm{T}} \cdots \boldsymbol{V}_{k-m}^{\mathrm{T}} \boldsymbol{H}_{k-m} \boldsymbol{V}_{k-m} \cdots \boldsymbol{V}_{k-2} \boldsymbol{V}_{k-1} \\
&+ \boldsymbol{V}_{k-1}^{\mathrm{T}} \boldsymbol{V}_{k-2}^{\mathrm{T}} \cdots \boldsymbol{V}_{k-m+1}^{\mathrm{T}} \boldsymbol{s}_{k-m} \rho_{k-m} \boldsymbol{s}_{k-m}^{\mathrm{T}} \boldsymbol{V}_{k-m+1} \cdots \boldsymbol{V}_{k-2} \boldsymbol{V}_{k-1} \\
&+ \cdots \\
&+ \boldsymbol{V}_{k-1}^{\mathrm{T}} \boldsymbol{s}_{k-2} \rho_{k-2} \boldsymbol{s}_{k-2}^{\mathrm{T}} \boldsymbol{V}_{k-1} \\
&+ \boldsymbol{s}_{k-1} \rho_{k-1} \boldsymbol{s}_{k-1}^{\mathrm{T}}
\end{aligned} \tag{6.40}$$

这样看起来非常复杂，但好处是我们只需要存储 $\boldsymbol{s}_{k-1}, \boldsymbol{s}_{k-2}, \cdots, \boldsymbol{s}_{k-m}$，$\boldsymbol{y}_{k-1}, \boldsymbol{y}_{k-2}, \cdots, \boldsymbol{y}_{k-m}$ 及 $\boldsymbol{H}_{k-m}$，就可以得到 $\boldsymbol{H}_k$。如果 $\boldsymbol{H}_{k-m}$ 是对角矩阵，那么原先存储 $\boldsymbol{H}_k$ 所需的 n^2 的空间可以降至 $(2m+1)n$。

尽管在理论上不如 BFGS 方法和 DFP 方法，但由于计算技巧上的改进，有限内存 BFGS 方法仍然是目前实际效果最好的拟牛顿法之一。

6.4 最小二乘

最小二乘是指形如

$$\min_{\boldsymbol{x}} \sum_{i=1}^{m} f_i(\boldsymbol{x})^2 \tag{6.41}$$

的问题。在实际问题中，最小二乘常常被作为一类模型来使用，也就是前面所讲过的代理模型法。在理想情况下，我们可能会要求所有的 $f_i(\boldsymbol{x})$ 等于 0，但实际上我们做不到这一点。一种解决方式就是求解问题(6.41)。

本节中我们总是假设 $m \geqslant n$，这是因为在 $m < n$ 时，$\sum_{i=1}^{m} f_i(\boldsymbol{x})^2 = 0$ 常常是有解的，也就没有必要去求最小二乘。

6.4.1 线性最小二乘

线性最小二乘是指 $f_i(\boldsymbol{x})$ 全都是线性函数。这时问题也可以写为

$$\min_{\boldsymbol{x}} \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|^2 \tag{6.42}$$

目标函数的梯度是 $2\boldsymbol{A}^{\mathrm{T}}(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b})$。因此我们可以令梯度等于零，得到方程组

$$\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}^{\mathrm{T}}\boldsymbol{b} \tag{6.43}$$

下面我们就来介绍如何求解式 (6.43)。

这时的一个选择是直接求解方程组 $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}^{\mathrm{T}}\boldsymbol{b}$。但这样的缺点是需要计算 $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}$，而矩阵乘法的计算量是比较大的。实际上常用的方法是对 $\boldsymbol{A}$ 进行 QR 分解，即 $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R}$，其中 $\boldsymbol{Q}$ 是正交矩阵（即 $\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{Q}$ 为单位矩阵），$\boldsymbol{R}$ 是上三角矩阵。假设我们已经有了这样一个分解，则代入式(6.43)后，我们可以得到

$$\boldsymbol{R}^{\mathrm{T}}\boldsymbol{R}\boldsymbol{x} = \boldsymbol{A}^{\mathrm{T}}\boldsymbol{b} \tag{6.44}$$

我们可以用前面介绍过的求解三角方程组的方法解出 $\boldsymbol{x}$，因此问题就集中在如何对 $\boldsymbol{A}$ 进行 QR 分解。

QR 分解的方法主要有三种：Householder 变换、Givens 变换和 Gram-Schmidt 正交化方法。矩阵分解并不是本章的重点，而介绍这三个方法会占用较大篇幅，有兴趣的读者请参考 [GVL12]。

6.4.2 非线性最小二乘

如果在最小二乘问题(6.41)中有某个 $f_i(\boldsymbol{x})$ 是非线性的，那么问题就变得困难很多。但是我们仍然可以使用导数优化方法来求解。

下面，我们介绍 Gauss Newton 法，这是一种迭代算法。我们记第 k 次迭代的迭代点为 $\boldsymbol{x}_k$，再定义 Jacobian 矩阵

$$\boldsymbol{J}(\boldsymbol{x}) = \begin{pmatrix} (\nabla f_1(\boldsymbol{x}))^{\mathrm{T}} \\ (\nabla f_2(\boldsymbol{x}))^{\mathrm{T}} \\ \vdots \\ (\nabla f_m(\boldsymbol{x}))^{\mathrm{T}} \end{pmatrix} \tag{6.45}$$

于是，目标函数 $\sum\limits_{i=1}^{m} f_i(\boldsymbol{x})^2$ 的梯度是

$$2\boldsymbol{J}^{\mathrm{T}}(\boldsymbol{x}) \begin{pmatrix} f_1(\boldsymbol{x}) \\ f_2(\boldsymbol{x}) \\ \vdots \\ f_m(\boldsymbol{x}) \end{pmatrix} \tag{6.46}$$

Hessian 矩阵是

$$2\boldsymbol{J}^{\mathrm{T}}(\boldsymbol{x})\boldsymbol{J}(\boldsymbol{x}) + 2\sum_{i=1}^{m} f_i(\boldsymbol{x})\nabla^2 f_i(\boldsymbol{x}) \tag{6.47}$$

我们近似地认为 Hessian 矩阵是 $2\boldsymbol{J}^{\mathrm{T}}(\boldsymbol{x})\boldsymbol{J}(\boldsymbol{x})$，则使用牛顿法可以得到迭代式

$$\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k - (\boldsymbol{J}^{\mathrm{T}}(\boldsymbol{x})\boldsymbol{J}(\boldsymbol{x}))^{-1}\boldsymbol{J}^{\mathrm{T}}(\boldsymbol{x}) \begin{pmatrix} f_1(\boldsymbol{x}) \\ f_2(\boldsymbol{x}) \\ \vdots \\ f_m(\boldsymbol{x}) \end{pmatrix} \tag{6.48}$$

这就是 Gauss-Newton 法的迭代格式。

如果用梯度下降法求解非线性最小二乘，则迭代格式是

$$x_{k+1} \leftarrow x_k - \lambda J^T(x) \begin{pmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_m(x) \end{pmatrix} \tag{6.49}$$

其中 λ 是某个步长。Levenberg[Lev44] 将梯度下降法和 Gauss-Newton 法融合，得到了一个新的迭代格式是

$$\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k - (\boldsymbol{J}^{\mathrm{T}}(\boldsymbol{x})\boldsymbol{J}(\boldsymbol{x}) + \lambda \boldsymbol{I})^{-1}\boldsymbol{J}^{\mathrm{T}}(\boldsymbol{x}) \begin{pmatrix} f_1(\boldsymbol{x}) \\ f_2(\boldsymbol{x}) \\ \vdots \\ f_m(\boldsymbol{x}) \end{pmatrix} \tag{6.50}$$

其中 λ 是一个参数。λ 为 0 时就是 Gauss-Newton 法，λ 很大时就会很接近梯度下降法。我们可以使用动态调整 λ 的策略。Marquardt[Mar63] 将式(6.50)中的单位矩阵替换为 Hessian 矩阵的对角线，将迭代格式变为

$$\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k - (\boldsymbol{J}^{\mathrm{T}}(\boldsymbol{x})\boldsymbol{J}(\boldsymbol{x}) + \lambda \mathrm{diag}(\boldsymbol{J}^{\mathrm{T}}(\boldsymbol{x})\boldsymbol{J}(\boldsymbol{x})))^{-1}\boldsymbol{J}^{\mathrm{T}}(\boldsymbol{x}) \begin{pmatrix} f_1(\boldsymbol{x}) \\ f_2(\boldsymbol{x}) \\ \vdots \\ f_m(\boldsymbol{x}) \end{pmatrix} \tag{6.51}$$

迭代格式(6.51)被称为 Levenberg-Marquardt 法。

第 7 章

迭代法

之前我们介绍了如何利用高斯消去法来求解线性方程组，这类方法属于直接法。本章我们要介绍如何利用迭代法求解线性方程组。当然，迭代法不仅可以用于求解线性方程组，也可以用于求解非线性方程组。与直接法相比，迭代法得到的解通常不够精确，但误差是可控的。使用迭代法可以得到误差容忍范围内的近似解，并且通过调整误差容忍值可以让迭代尽早终止，从而提高计算效率。本章我们除了介绍几种流行的迭代法的算法流程，还会仔细分析这些方法的收敛条件和有效性。

7.1 线性方程组的迭代法

7.1.1 一阶定常格式迭代法

迭代法是指通过近似解序列 $\boldsymbol{x}^{(0)}, \boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(k)}, \cdots$ 不断逼近准确解 $\boldsymbol{x}_*$。如果相邻的两个近似解满足一个固定的函数关系，则称其为固定格式迭代法。在求解线性方程组时，这种固定的迭代格式可以写为

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{G}\boldsymbol{x}^{(k)} + \boldsymbol{g} \tag{7.1}$$

这里的 $\boldsymbol{x}^{(k+1)}$、$\boldsymbol{x}^{(k)}$ 和 $\boldsymbol{g}$ 都是向量，$\boldsymbol{G}$ 为方阵，我们也把它叫作一阶定常格式迭代法。“一阶”是指迭代格式只用到了上一步的近似解；“定常”是指矩阵 $\boldsymbol{G}$ 是固定的，不随迭代次数改变。精确解 $\boldsymbol{x}_*$ 应当满足 $\boldsymbol{x}_* = \boldsymbol{G}\boldsymbol{x}_* + \boldsymbol{g}$，即 $(\boldsymbol{I} - \boldsymbol{G})\boldsymbol{x}_* = \boldsymbol{g}$。所以我们一般要求

$\boldsymbol{I}-\boldsymbol{G}$ 非奇异，以保证存在精确解。迭代收敛是指当 $k\to+\infty$ 时，$\boldsymbol{x}^{(k)}\to\boldsymbol{x}_*$，也就是误差 $\boldsymbol{e}^{(k)}=\boldsymbol{x}^{(k)}-\boldsymbol{x}_*\to 0$。不难推导出误差 $\boldsymbol{e}^{(k)}$ 关于初始误差的递推关系为

$$\boldsymbol{e}^{(k)}=\boldsymbol{G}^k\boldsymbol{e}^{(0)} \tag{7.2}$$

定理 7.1 当且仅当 $\boldsymbol{G}$ 的谱半径（Spectral Norm）小于 1，即矩阵 $\boldsymbol{G}$ 的最大奇异值小于 1 时，迭代法才会收敛。

证 根据谱半径的定义，可以得到 $\|\boldsymbol{G}\boldsymbol{x}\|_2\leqslant\rho(\boldsymbol{G})\|\boldsymbol{x}\|_2$。于是我们有

$$\|\boldsymbol{e}^{(k)}\|_2=\|\boldsymbol{G}^k\boldsymbol{e}^{(0)}\|_2\leqslant\rho(\boldsymbol{G})^k\|\boldsymbol{e}^{(0)}\|_2 \tag{7.3}$$

当 $\rho(\boldsymbol{G})<1$ 且 $k\to+\infty$ 时，$\rho(\boldsymbol{G})^k\to 0$。所以 $\|\boldsymbol{e}^{(k)}\|_2\to 0$。 □

因为谱半径和诱导范数（Induced Norm）之间都满足 $\rho(\boldsymbol{G})\leqslant\|\boldsymbol{G}\|$，所以如果能够找到矩阵的某一诱导范数小于 1，则也能证明迭代法收敛。接下来我们要介绍三种常见的一阶定常迭代：Jacobi 迭代、Gauss-Seidel 迭代和超松弛迭代（SOR）。

1. Jacobi 迭代

我们把线性方程组展开来写：

$$\begin{cases} a_{1,1}x_1+a_{1,2}x_2+\cdots+a_{1,n}x_n &= b_1\\ a_{2,1}x_1+a_{2,2}x_2+\cdots+a_{2,n}x_n &= b_2\\ &\vdots\\ a_{n,1}x_1+a_{n,2}x_2+\cdots+a_{n,n}x_n &= b_n \end{cases} \tag{7.4}$$

假设 $\forall i=1,2,\cdots,n$，满足 $a_{i,i}\neq 0$，则我们可以有如下变换：

$$\begin{cases} x_1 &= -\dfrac{1}{a_{1,1}}(a_{1,2}x_2+\cdots+a_{1,n}x_n)+\dfrac{b_1}{a_{1,1}}\\ x_2 &= -\dfrac{1}{a_{2,2}}(a_{2,1}x_1+a_{2,3}x_3+\cdots+a_{2,n}x_n)+\dfrac{b_2}{a_{2,2}}\\ &\vdots\\ x_n &= -\dfrac{1}{a_{n,n}}(a_{n,1}x_1+a_{n,2}x_2+\cdots+a_{n,n-1}x_{n-1})+\dfrac{b_n}{a_{n,n}} \end{cases} \tag{7.5}$$

如果将等号右边都用上一次迭代的近似解，等号左边都用新的迭代近似解，那么可以得到如下迭代格式：

$$\begin{cases} x_1^{(k+1)} &= -\dfrac{1}{a_{1,1}}(a_{1,2}x_2^{(k)} + \cdots + a_{1,n}x_n^{(k)}) + \dfrac{b_1}{a_{1,1}} \\ x_2^{(k+1)} &= -\dfrac{1}{a_{2,2}}(a_{2,1}x_1^{(k)} + a_{2,3}x_3^{(k)} + \cdots + a_{2,n}x_n^{(k)}) + \dfrac{b_2}{a_{2,2}} \\ &\vdots \\ x_n^{(k+1)} &= -\dfrac{1}{a_{n,n}}(a_{n,1}x_1^{(k)} + a_{n,2}x_2^{(k)} + \cdots + a_{n,n-1}x_{n-1}^{(k)}) + \dfrac{b_n}{a_{n,n}} \end{cases} \tag{7.6}$$

下面我们将把上式简写成矩阵形式。设 $\boldsymbol{D}$、$\boldsymbol{L}$、$\boldsymbol{U}$ 分别为对角阵、下三角及上三角阵。

$$\boldsymbol{D} = \begin{pmatrix} a_{1,1} & 0 & \cdots & 0 & 0 \\ 0 & a_{2,2} & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{n-1,n-1} & 0 \\ 0 & 0 & \cdots & 0 & a_{n,n} \end{pmatrix} \tag{7.7}$$

$$\boldsymbol{L} = \begin{pmatrix} 0 & 0 & \cdots & 0 & 0 \\ -a_{2,1} & 0 & \cdots & 0 & 0 \\ -a_{3,1} & -a_{3,2} & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ -a_{n,1} & -a_{n,2} & \cdots & -a_{n,n-1} & 0 \end{pmatrix} \tag{7.8}$$

$$\boldsymbol{U} = \begin{pmatrix} 0 & -a_{1,2} & -a_{1,3} & \cdots & -a_{1,n} \\ 0 & 0 & -a_{2,3} & \cdots & -a_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & -a_{n-1,n} \\ 0 & 0 & 0 & \cdots & 0 \end{pmatrix} \tag{7.9}$$

如果满足 $\boldsymbol{A} = \boldsymbol{D} - \boldsymbol{L} - \boldsymbol{U}$，其中 $\boldsymbol{A}$ 为原问题稀疏矩阵，则迭代格式可以改写为

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{D}^{-1}(\boldsymbol{L} + \boldsymbol{U})\boldsymbol{x}^{(k)} + \boldsymbol{D}^{-1}\boldsymbol{b} \tag{7.10}$$

它的收敛条件是 $\boldsymbol{D}^{-1}(\boldsymbol{L}+\boldsymbol{U})$ 的谱半径小于 1，这个条件显然是不容易验证的。在实际应用时，我们经常使用一个容易验证的充分条件来代替谱半径小于 1：原问题系数矩阵 $\boldsymbol{A}$ 满足行对角占优，即每一个对角线元素都大于同行所有其他元素之和；或者列对角占优，即每一个对角线元素都大于同列所有其他元素之和。

2. Gauss-Seidel 迭代

从 Jacobi 迭代格式的推导过程中，我们会产生一个很自然的想法，在计算 $x_i^{(k+1)}$ 时，既然 $\forall j<i$，已经得到了第 $k+1$ 次更新的结果，那么就可以在迭代式中使用了。具体形式是

$$\begin{cases} x_1^{(k+1)} &= -\dfrac{1}{a_{1,1}}(a_{1,2}x_2^{(k)}+\cdots+a_{1,n}x_n^{(k)})+\dfrac{b_1}{a_{1,1}} \\ x_2^{(k+1)} &= -\dfrac{1}{a_{2,2}}(a_{2,1}x_1^{(k+1)}+a_{2,3}x_3^{(k)}+\cdots+a_{2,n}x_n^{(k)})+\dfrac{b_2}{a_{2,2}} \\ &\vdots \\ x_n^{(k+1)} &= -\dfrac{1}{a_{n,n}}(a_{n,1}x_1^{(k+1)}+a_{n,2}x_2^{(k+1)}+\cdots+a_{n,n-1}x_{n-1}^{(k+1)})+\dfrac{b_n}{a_{n,n}} \end{cases} \tag{7.11}$$

写成方程的形式是

$$\boldsymbol{x}^{(k+1)}=\boldsymbol{D}^{-1}(\boldsymbol{L}\boldsymbol{x}^{(k+1)}+\boldsymbol{U}\boldsymbol{x}^{(k)})+\boldsymbol{D}^{-1}\boldsymbol{b} \tag{7.12}$$

整理后得到如下形式：

$$\boldsymbol{x}^{(k+1)}=(\boldsymbol{D}-\boldsymbol{L})^{-1}\boldsymbol{U}\boldsymbol{x}^{(k)}+(\boldsymbol{D}-\boldsymbol{L})^{-1}\boldsymbol{b} \tag{7.13}$$

收敛条件是 $(\boldsymbol{D}-\boldsymbol{L})^{-1}\boldsymbol{U}$ 的谱半径小于 1，一个充分条件也是原问题系数矩阵是行或列对角占优的。

Jacobi 和 Gauss-Seidel 迭代的收敛性的充要条件是对应迭代格式的定常矩阵的谱半径小于 1，但它们的迭代矩阵是不同的。原问题系数矩阵 $\boldsymbol{A}$ 对角占优是这两种方法收敛的充分条件。除此之外，我们关心的问题如下。

- Gauss-Seidel 迭代收敛是 Jacobi 迭代收敛的充分条件吗？
- Jacobi 迭代收敛是 Gauss-Seidel 迭代收敛的充分条件吗？
- 当两种迭代都收敛时，哪种迭代的收敛速度更快？

事实上，当 Gauss-Seidel 迭代收敛时，Jacobi 迭代可能会发散；当 Jacobi 迭代收敛时，Gauss-Seidel 迭代可能会发散。也就是说，它们的收敛关系互相都不是充分条件。比如考虑如下问题：

$$\begin{pmatrix} 1 & 2 & -2 \\ 1 & 1 & 1 \\ 2 & 2 & 1 \end{pmatrix} \boldsymbol{x} = \begin{pmatrix} 1 \\ 3 \\ 5 \end{pmatrix} \tag{7.14}$$

我们很容易验证线性方程组 (7.14) 的唯一解是 $\boldsymbol{x}_* = (1,1,1)^{\mathrm{T}}$。表7.1是使用 Jacobi 迭代和 Gauss-Seidel 迭代的近似解序列，初始解 $\boldsymbol{x}_0$ 设为 $(0.8,0.8,0.8)^{\mathrm{T}}$。Jacobi 迭代经过短短四五步便已经收敛到准确解。在 Gauss-Seidel 迭代中虽然我们只列出了 6 步的结果，但我们可以发现这种发散的趋势。事实上当我们采用 Gauss-Seidel 迭代更多步后甚至还会出现越界。可以发现，Gauss-Seidel 迭代发散，而 Jacobi 迭代收敛。

表 7.1 方程组 (7.14) 的 Jacobi 迭代和 Gauss-Seidel 迭代近似解序列

方 法	近似解序列	$\|\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)}\|_2$
Jacobi	$(0.8,0.8,08)^{\mathrm{T}}$	–
	$(1.0,1.4,1.8)^{\mathrm{T}}$	1.183
	$(1.8,0.2,0.2)^{\mathrm{T}}$	2.154
	$(1,1,1)^{\mathrm{T}}$	1.385
	$(1,1,1)^{\mathrm{T}}$	0
Gauss-Seidel	$(0.8,0.8,0.8)^{\mathrm{T}}$	–
	$(1,1.2,0.6)^{\mathrm{T}}$	0.490
	$(-0.2,2.6,0.2)^{\mathrm{T}}$	1.887
	$(-3.8,6.6,-0.6)^{\mathrm{T}}$	5.441
	$(-13.4,17.0,-2.2)^{\mathrm{T}}$	14.244
	$(-37.4,42.6,-5.4)^{\mathrm{T}}$	35.236
	⋮	⋮

我们再来考虑另一个问题：

$$\begin{pmatrix} 1 & 0.6 & 0.6 \\ 0.6 & 1 & 0.6 \\ 0.6 & 0.6 & 1 \end{pmatrix} \boldsymbol{x} = \begin{pmatrix} 2.2 \\ 2.2 \\ 2.2 \end{pmatrix} \tag{7.15}$$

同样，我们也很容易验证线性方程组 (7.15) 的唯一解是 $\boldsymbol{x}_* = (1,1,1)^{\mathrm{T}}$。表7.2是分别使用 Jacobi 迭代和 Gauss-Seidel 迭代的近似解序列，初始解 $\boldsymbol{x}_0$ 设为 $(0.8,0.8,0.8)^{\mathrm{T}}$。可以发现，Jacobi 迭代发散，而 Gauss-Seidel 迭代收敛。

通过举反例的方法，我们可以很容易地验证两种迭代法互相没有充分性。

表 7.2 方程组 (7.15) 的 Jacobi 迭代和 Gauss-Seidel 迭代的近似解序列

方 法	步 数 k	近似解序列	$\|\boldsymbol{x}^{(k+1)}-\boldsymbol{x}^{(k)}\|_2$
Jacobi	1	$(0.8,0.8,0.8)^{\mathrm{T}}$	–
	2	$(1.24,1.24,1.24)^{\mathrm{T}}$	0.915
	3	$(0.712,0.712,0.712)^{\mathrm{T}}$	1.097
	⋮	⋮	⋮
	40	$(245.9619,245.9619,245.9619)^{\mathrm{T}}$	933.430
	⋮	⋮	⋮
	100	$(13802997,13802997,13802997)^{\mathrm{T}}$	52596477.872
	⋮	⋮	⋮
Gauss-Seidel	1	$(0.8,0.8,0.8)^{\mathrm{T}}$	–
	2	$(1.240,0.976,0.870)^{\mathrm{T}}$	0.479
	3	$(1.092,1.022,0.931)^{\mathrm{T}}$	0.166
	⋮	⋮	⋮
	14	$(1.0000024,0.9999813,1.0000098)^{\mathrm{T}}$	2.949×10^{-5}
	⋮	⋮	⋮
	25	$(1,1,1)^{\mathrm{T}}$	6.401×10^{-9}

当原问题系数矩阵 A 是严格对角占优时，两种迭代法都收敛，此时 Gauss-Seidel 迭代的收敛速度要比 Jacobi 迭代的收敛速度更快。对于这方面的证明，感兴趣的读者可以查找更多的文献。我们通过一个例子来验证这个结论，考虑如下问题：

$$\begin{pmatrix} 1 & a & a \\ a & 1 & a \\ a & a & 1 \end{pmatrix} \boldsymbol{x} = \begin{pmatrix} 1+2a \\ 1+2a \\ 1+2a \end{pmatrix} \tag{7.16}$$

那么无论 a 取何值，只要原问题存在唯一解，则解为 $(1,1,1)^{\mathrm{T}}$。很明显，当 $-0.5<a<0.5$ 时，系数矩阵是严格对角占优的。我们取 $a=\pm0.4,\pm0.3,\pm0.2,\pm0.1$，得到它们的迭代步数对比如表7.3所示。当然，这些例子只能用于观察特定问题的收敛速度比较，要得出确定的结论还要有更严谨的数学证明。

表 7.3 方程组 (7.16) 的迭代法收敛比较

a 的取值	Jacobi 迭代步数	Gauss-Seidel 迭代步数
0.4	82	15
0.3	36	12
0.2	21	10

续表

a 的取值	Jacobi 迭代步数	Gauss-Seidel 迭代步数
0.1	12	8
-0.1	12	9
-0.2	20	13
-0.3	34	20
-0.4	72	40

3. 超松弛迭代

超松弛迭代在 Gauss-Seidel 迭代的基础上引入了松弛因子 ω。用 Gauss-Seidel 法更新 $x_i^{(k+1)}$ 后，我们把它与 $x_i^{(k)}$ 取线性组合作为修正，得到 SOR 的基本形式：

$$\boldsymbol{x}^{(k+1)} = \omega(\boldsymbol{D}^{-1}(\boldsymbol{L}\boldsymbol{x}^{(k+1)} + \boldsymbol{U}\boldsymbol{x}^{(k)}) + \boldsymbol{D}^{-1}\boldsymbol{b}) + (1-\omega)\boldsymbol{x}^{(k)} \tag{7.17}$$

将具体递推形式代入后得到的最终形式是：

$$\boldsymbol{x}^{(k+1)} = (\boldsymbol{I} - \omega\boldsymbol{D}^{-1}\boldsymbol{L})^{-1}((1-\omega)\boldsymbol{I} + \omega\boldsymbol{D}^{-1}\boldsymbol{U})\boldsymbol{x}^{(k)} + \omega(\boldsymbol{I} - \omega\boldsymbol{D}^{-1}\boldsymbol{L})^{-1}\boldsymbol{D}^{-1}\boldsymbol{b} \tag{7.18}$$

如果 $\omega = 1$，那么它就是 Gauss-Seidel 迭代；如果 $\omega = 0$，那么迭代是没有意义的，因为它并不会更新解序列。当 $\omega > 1$ 时，我们称之为超松弛迭代法（SOR）。如果 SOR 方法的收敛等价于迭代格式中的定常矩阵的谱半径小于 1，那么这个定常矩阵的行列式的绝对值必须小于 1，因为行列式等于所有特征值的积。再根据三角矩阵的行列式等于其对角线上元素的乘积，得出

$$\begin{aligned} &\det((\boldsymbol{I} - \omega\boldsymbol{D}^{-1}\boldsymbol{L})^{-1}((1-\omega)\boldsymbol{I} + \omega\boldsymbol{D}^{-1}\boldsymbol{U})) \\ &= \det((\boldsymbol{I} - \omega\boldsymbol{D}^{-1}\boldsymbol{L})^{-1})\det((1-\omega)\boldsymbol{I} + \omega\boldsymbol{D}^{-1}\boldsymbol{U}) \\ &= (1-\omega)^n \end{aligned} \tag{7.19}$$

所以 SOR 方法收敛的一个必要条件就是 $|(1-\omega)^n| < 1$，即 $0 < \omega < 2$。

7.1.2 Krylov 子空间算法

在之前介绍的定常迭代算法中，我们提到这些方法并不非常实用，因为它们的收敛条件较为苛刻，而很多实际问题并不满足。目前求解大规模稀疏线性方程组问题的最普遍的

方法是 Krylov 子空间算法，其基本思想是在一个较小的子空间 $\mathcal{K} \subset \mathbb{R}^n$ 中找出原问题的近似解，而我们知道这个近似解就是准确解在子空间 $\mathcal{K}$ 上的投影，因此这类方法也是投影算法。投影算法的一般形式是：找到近似解 $\hat{\boldsymbol{x}} \in \mathcal{K}$，使

$$\boldsymbol{b} - \boldsymbol{A}\hat{\boldsymbol{x}} \perp \mathcal{L} \tag{7.20}$$

其中 $\mathcal{L}$ 是另外一个 m 维的线性空间。设 $\{v_1, v_2, \cdots, v_m\}$ 和 $\{w_1, w_2, \cdots, w_m\}$ 分别是 $\mathcal{K}$ 和 $\mathcal{L}$ 的标准正交基。令矩阵 $\boldsymbol{V} = (v_1, v_2, \cdots, v_m)$，矩阵 $\boldsymbol{W} = (w_1, w_2, \cdots, w_m)$，由于 $\hat{\boldsymbol{x}} \in \boldsymbol{x}^{(0)} + \mathcal{K}$，因此存在 $\boldsymbol{y} \in \mathcal{R}^m$，使得 $\hat{\boldsymbol{x}} = \boldsymbol{x}^{(0)} + \boldsymbol{V}\boldsymbol{y}$。由正交性条件 (7.20) 可得

$$\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(0)} - \boldsymbol{A}\boldsymbol{V}\boldsymbol{y} \perp w_i, \forall i = 1, 2, \cdots, m \tag{7.21}$$

即

$$\boldsymbol{W}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{V}\boldsymbol{y} = \boldsymbol{W}^{\mathrm{T}}(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(0)}) \tag{7.22}$$

为了表述方便，对于任意 i，我们称 $\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(i)}$ 为残差，并记作 $\boldsymbol{r}^{(i)}$。如果矩阵 $\boldsymbol{W}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{V}$ 可逆，则 $\hat{\boldsymbol{x}}$ 有最佳近似解的解析形式：

$$\hat{\boldsymbol{x}} = \boldsymbol{x}_0 + \boldsymbol{V}(\boldsymbol{W}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{V})^{-1}\boldsymbol{W}^{\mathrm{T}}\boldsymbol{r}_0 \tag{7.23}$$

下面我们通过若干概念逐步介绍 Krylov 子空间算法的思想及其变种。

1. 投影算子

投影算子首先是一个线性算子，设 L 是空间 X 到空间 Y 的线性算子，则它满足：

- $L(\alpha\boldsymbol{x}) = \alpha L(\boldsymbol{x}), \forall \boldsymbol{x} \in X, \alpha \in \mathbb{R}$
- $L(\boldsymbol{x}_1 + \boldsymbol{x}_2) = L(\boldsymbol{x}_1) + L(\boldsymbol{x}_2), \forall \boldsymbol{x}_1, \boldsymbol{x}_2 \in X$

L 的值域空间定义为：

$$\mathcal{R}(L) = \{L(\boldsymbol{x}) \in Y : \boldsymbol{x} \in X\} \tag{7.24}$$

L 的零空间又称为核，其定义为：

$$\mathcal{N}(L) = \{\boldsymbol{x} \in X : L(\boldsymbol{x}) = 0\} \tag{7.25}$$

$\mathcal{R}(L)$ 是 Y 空间的子空间，而 $\mathcal{N}(L)$ 是 X 空间的子空间。如果 P 是一个线性算子，且满足

$$P^2 = P \tag{7.26}$$

则我们称 P 为投影算子或者投影变换，也可以简称为投影。如果这时还满足 $\mathcal{R}(P) \perp \mathcal{N}(P)$，即 $\forall \boldsymbol{x} \in \mathcal{R}(P)$，$\boldsymbol{y} \in \mathcal{N}(P)$，都有 $(\boldsymbol{x}, \boldsymbol{y}) = 0$，则我们把 P 叫作正交投影，其他投影算子就对应为斜投影算子。图7.1给出了二维空间中点 x 在向量 $\boldsymbol{A}$ 上的正交投影，以及三维空间中点 x 在平面 $\boldsymbol{A}$ 上的正交投影。

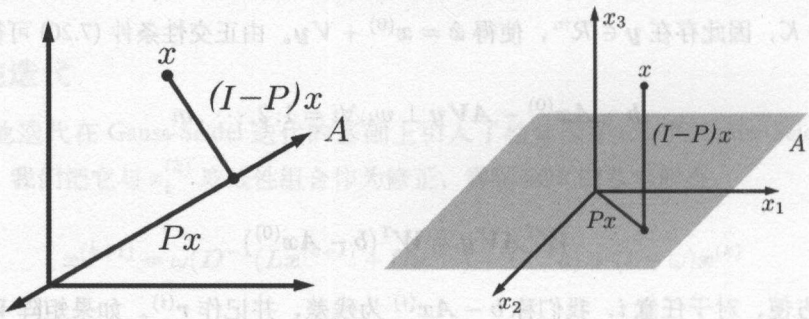


图 7.1　二维和三维实数空间中的正交投影图示

正交投影算子 P 可以表示成一个 $n \times n$ 的矩阵，但表示形式并不唯一。

2. Krylov 子空间

设 $\boldsymbol{A} \in \mathbb{R}^{n \times n}$，$\boldsymbol{r} \in \mathbb{R}^n$，则我们称

$$\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}) \triangleq \text{span}\{\boldsymbol{r}, \boldsymbol{A}\boldsymbol{r}, \cdots, \boldsymbol{A}^{m-1}\boldsymbol{r}\} \subseteq \mathbb{R}^n \tag{7.27}$$

为由 $\boldsymbol{A}$ 和 $\boldsymbol{r}$ 生成的 Krylov 子空间，简记为 $\mathcal{K}_m$，它具有的性质如下。

- Krylov 子空间的嵌套性：$\mathcal{K}_1 \subseteq \mathcal{K}_2 \subseteq \cdots \subseteq \mathcal{K}_m$。
- $\mathcal{K}_m$ 的维数不超过 m。
- $\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}) = \{\boldsymbol{x} = p(\boldsymbol{A})\boldsymbol{r} : p\text{为次数小于 } m \text{ 的多项式}\}$。

Krylov 子空间方法就是在 Krylov 子空间中寻找近似解。一个首先要问的问题就是：线性方程组 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ 的解与 Krylov 子空间的关系。显然存在某个 m 使得 $\mathcal{K}_{m+1}(\boldsymbol{A}, \boldsymbol{r}) = \mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r})$，也就是存在 $c_0, c_1, \ldots, c_{m-1}$，使得

$$\boldsymbol{A}^m\boldsymbol{r} = \sum_{i=0}^{m-1} c_i \boldsymbol{A}^i \boldsymbol{r} \tag{7.28}$$

假设 m 是满足式 (7.28) 的最小的一个，这样一定有 $c_0 \neq 0$，否则有

$$\boldsymbol{A}^{m-1}\boldsymbol{r} = \sum_{i=0}^{m-2} c_{i+1} \boldsymbol{A}^i \boldsymbol{r} \tag{7.29}$$

成立，与假设矛盾。既然 $c_0 \neq 0$，则我们可以将式 (7.28) 改写为

$$\boldsymbol{A}\left(\frac{\boldsymbol{A}^{m-1}\boldsymbol{r} - \sum_{i=1}^{m-1} c_i \boldsymbol{A}^{i-1}\boldsymbol{r}}{c_0}\right) = \boldsymbol{r} \tag{7.30}$$

如果取 $\boldsymbol{r} = \boldsymbol{b}$，则

$$\boldsymbol{x} = \frac{\boldsymbol{A}^{m-1}\boldsymbol{r} - \sum_{i=1}^{m-1} c_i \boldsymbol{A}^{i-1}\boldsymbol{r}}{c_0} \tag{7.31}$$

就是方程组 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ 的解。如果取 $\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$，其中 $\boldsymbol{x}_0$ 为任意向量，则

$$\boldsymbol{x} = \frac{\boldsymbol{A}^{m-1}\boldsymbol{r} - \sum_{i=1}^{m-1} c_i \boldsymbol{A}^{i-1}\boldsymbol{r}}{c_0} + \boldsymbol{x}_0 \tag{7.32}$$

就是方程组 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ 的解，因此求出式 (7.28) 就意味着找到了方程组的解。

3. Arnoldi 算法

Arnoldi 算法是 1951 年由 [Arn51] 在 Krylov 子空间的思想基础上提出的。理论上，直接计算 Krylov 子空间就可以得到方程组的解了，但实际上计算中的数值的稳定性会很差，一个解决方案就是计算 Krylov 子空间的单位正交基底，我们可以使用 Gram-Schmidt 正交化来完成。假设 $\mathcal{K}_j(\boldsymbol{A}, \boldsymbol{r})$ 的一组单位正交基底是 $\{\boldsymbol{q}_1, \boldsymbol{q}_2, \cdots, \boldsymbol{q}_j\}$，要计算 $\mathcal{K}_{j+1}(\boldsymbol{A}, \boldsymbol{r})$ 的基底，则只需处理新增的向量 $\boldsymbol{A}^j\boldsymbol{r}$，扣除它在正交基底上的投影，即

$$\tilde{\boldsymbol{q}}_{j+1} = \boldsymbol{A}^j\boldsymbol{r} - \sum_{i=1}^{j} (\boldsymbol{q}_i^{\mathrm{T}} \boldsymbol{A}^j \boldsymbol{v}) \boldsymbol{q}_i \tag{7.33}$$

新的正交基底即 $\boldsymbol{q}_{j+1} = \dfrac{\tilde{\boldsymbol{q}}_{j+1}}{\|\tilde{\boldsymbol{q}}_{j+1}\|}$。

我们可以使用数学归纳法证明

$$\mathcal{K}_{j+1}(\boldsymbol{A},\boldsymbol{r}) = \text{span}\{\boldsymbol{q}_1,\boldsymbol{q}_2,\cdots,\boldsymbol{q}_j,\boldsymbol{A}\boldsymbol{q}_j\} \tag{7.34}$$

因此，$\boldsymbol{A}\boldsymbol{q}_j$ 可以用于替代 $\boldsymbol{A}^j\boldsymbol{v}$，即

$$\tilde{\boldsymbol{q}}_{j+1} = \boldsymbol{A}\boldsymbol{q}_j - \sum_{i=1}^{j}(\boldsymbol{q}_i^{\mathrm{T}}\boldsymbol{A}\boldsymbol{q}_j)\boldsymbol{q}_i \tag{7.35}$$

下面我们用矩阵表示 Arnoldi 算法，便于后面进一步讨论。式 (7.35) 可以改写为

$$\begin{aligned}\boldsymbol{A}\boldsymbol{q}_j &= \sum_{i=1}^{j}(\boldsymbol{q}_i^{\mathrm{T}}\boldsymbol{A}\boldsymbol{q}_j)\boldsymbol{q}_i + \tilde{\boldsymbol{q}}_{j+1} \\ &= \sum_{i=1}^{j}(\boldsymbol{q}_i^{\mathrm{T}}\boldsymbol{A}\boldsymbol{q}_j)\boldsymbol{q}_i + \|\tilde{\boldsymbol{q}}_{j+1}\|\boldsymbol{q}_{j+1}\end{aligned} \tag{7.36}$$

因此我们有

$$\begin{aligned}\boldsymbol{A}\boldsymbol{Q}_j &= (\boldsymbol{A}\boldsymbol{q}_1,\boldsymbol{A}\boldsymbol{q}_2,\cdots,\boldsymbol{A}\boldsymbol{q}_{j-1},\boldsymbol{A}\boldsymbol{q}_j) \\ &= (\boldsymbol{q}_1,\boldsymbol{q}_2,\cdots,\boldsymbol{q}_j,\boldsymbol{q}_{j+1})\begin{pmatrix} \boldsymbol{q}_1^{\mathrm{T}}\boldsymbol{A}\boldsymbol{q}_1 & \boldsymbol{q}_1^{\mathrm{T}}\boldsymbol{A}\boldsymbol{q}_2 & \cdots & \boldsymbol{q}_1^{\mathrm{T}}\boldsymbol{A}\boldsymbol{q}_{j-1} & \boldsymbol{q}_1^{\mathrm{T}}\boldsymbol{A}\boldsymbol{q}_j \\ \|\tilde{\boldsymbol{q}}_2\| & \boldsymbol{q}_2^{\mathrm{T}}\boldsymbol{A}\boldsymbol{q}_2 & \cdots & \boldsymbol{q}_2^{\mathrm{T}}\boldsymbol{A}\boldsymbol{q}_{j-1} & \boldsymbol{q}_2^{\mathrm{T}}\boldsymbol{A}\boldsymbol{q}_j \\ 0 & \|\tilde{\boldsymbol{q}}_3\| & \cdots & \boldsymbol{q}_3^{\mathrm{T}}\boldsymbol{A}\boldsymbol{q}_{j-1} & \boldsymbol{q}_3^{\mathrm{T}}\boldsymbol{A}\boldsymbol{q}_j \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \|\tilde{\boldsymbol{q}}_j\| & \boldsymbol{q}_{j-1}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{q}_j \\ 0 & 0 & \cdots & 0 & \|\tilde{\boldsymbol{q}}_{j+1}\| \end{pmatrix} \\ &\triangleq \boldsymbol{Q}_{j+1}\boldsymbol{H}_j\end{aligned} \tag{7.37}$$

4. 广义极小化残差方法

广义极小化残差方法（GMRES）[SS86] 就是要求解优化问题：

$$\begin{aligned}\min_{\boldsymbol{z}} \quad & \|\boldsymbol{A}(\boldsymbol{x}_0+\boldsymbol{z})-\boldsymbol{b}\| \\ \text{s.t.} \quad & \boldsymbol{z}\in\mathcal{K}_j(\boldsymbol{A},\boldsymbol{r}_0)\end{aligned} \tag{7.38}$$

由于 $\mathcal{K}_j(\boldsymbol{A},\boldsymbol{r}_0) = \text{span}(\boldsymbol{q}_1,\boldsymbol{q}_2,\cdots,\boldsymbol{q}_j)$，所以上述问题等价于

$$\min_{\boldsymbol{y}} \quad \|\boldsymbol{A}(\boldsymbol{x}_0+\boldsymbol{Q}_j\boldsymbol{y})-\boldsymbol{b}\| \tag{7.39}$$

根据前面的 Arnoldi 算法，我们可以将目标函数改写为

$$\begin{aligned} \|\boldsymbol{A}(\boldsymbol{x}_0+\boldsymbol{Q}_j\boldsymbol{y})-\boldsymbol{b}\|^2 &= \|\boldsymbol{A}\boldsymbol{Q}_j\boldsymbol{y}-\boldsymbol{r}_0\|^2 \\ &= \|\boldsymbol{A}\boldsymbol{Q}_j\boldsymbol{y}-\|\boldsymbol{r}_0\|\boldsymbol{q}_1\|^2 \\ &= \|\boldsymbol{Q}_{j+1}(\boldsymbol{H}_j\boldsymbol{y}-\|\boldsymbol{r}_0\|\boldsymbol{e}_1)\|^2 \\ &= \|\boldsymbol{H}_j\boldsymbol{y}-\|\boldsymbol{r}_0\|\boldsymbol{e}_1\|^2 \end{aligned} \tag{7.40}$$

其中 $\boldsymbol{e}_1$ 是第 1 个分量为 1、其余分量都为 0 的向量。要极小化 $\|\|\boldsymbol{r}_0\|\boldsymbol{e}_1-\boldsymbol{H}_j\boldsymbol{y}\|$ 就是要求 $\boldsymbol{H}_j\boldsymbol{y}=\|\boldsymbol{r}_0\|\boldsymbol{e}_1$ 的最小二乘解，当然，我们可以采用前面介绍过的方法。这里 $\boldsymbol{H}_j$ 实际上非常接近上三角矩阵，所以可以采用 Givens 变换将其变为上三角矩阵，然后直接通过代入法得到解。

7.1.3 无约束优化方法

除了使用之前介绍的固定格式迭代法和 Krylov 子空间方法，如果系数矩阵 $\boldsymbol{A}$ 是对称正定的，则我们考虑以下函数的最小值：

$$f(\boldsymbol{x})=\frac{1}{2}\boldsymbol{x}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b}^{\mathrm{T}}\boldsymbol{x} \tag{7.41}$$

由于 $\boldsymbol{A}$ 是对称正定的，函数 $f(\boldsymbol{x})$ 实际上是严格凸函数，那么它的最值出现在 $\nabla f(\boldsymbol{x})=0$ 处，即 $\boldsymbol{A}\boldsymbol{x}=\boldsymbol{b}$ 处。对于这样的无约束优化问题，可以参考本书前面章节中介绍过的方法，比如共轭梯度法，它在求解稀疏矩阵时有很广泛的应用。这类迭代法的迭代格式为：

$$\boldsymbol{x}^{(k+1)}=\boldsymbol{x}^{(k)}+\alpha^{(k)}\boldsymbol{d}^{(k)} \tag{7.42}$$

其中 $\alpha^{(k)}$ 是第 $k+1$ 次迭代的搜索步长，$\boldsymbol{d}^{(k)}$ 是第 $k+1$ 次迭代的搜索方向。但是我们知道共轭梯度法适用的是系数矩阵对称正定的情况，如果只是正定，则当然还可以使用双共轭梯度法，但对于更一般的情形，这些方法不能保证收敛。

7.2 非线性方程组的迭代法

非线性方程组的一般形式为

$$\begin{cases} f_1(x_1, x_2, \cdots, x_n) &= 0 \\ f_2(x_1, x_2, \cdots, x_n) &= 0 \\ & \vdots \\ f_n(x_1, x_2, \cdots, x_n) &= 0 \end{cases} \tag{7.43}$$

设 $\boldsymbol{x} = (x_1, x_2, \cdots, x_n)^{\mathrm{T}}$，$F(\boldsymbol{x}) = (f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \cdots, f_n(\boldsymbol{x}))^{\mathrm{T}}$，则方程可以改写为 $F(\boldsymbol{x}) = 0$。当问题只有一个未知量时，问题会退化为非线性方程。我们将介绍用于求解非线性方程组的不动点迭代法和 Newton-Raphson 迭代法。

7.2.1 不动点迭代

不动点迭代的基本思想是通过构造与原问题 $F(\boldsymbol{x}) = 0$ 等价的新问题 $\boldsymbol{x} = \Phi(\boldsymbol{x})$，利用迭代格式 $\boldsymbol{x}^{(k+1)} = \Phi(\boldsymbol{x}^{(k)})$ 来得到近似解。我们不妨举例来说明这个过程。

引入以下非线性方程组作为要求解的问题：

$$\begin{cases} x_1 + x_2 - 3 = 0 \\ 3x_1^2 + 2x_2^2 - 11 = 0 \end{cases} \tag{7.44}$$

非线性方程组的解可能并不唯一，所以对于这个问题，我们把问题限定于只需要找到从 $\boldsymbol{x}_0 = (0.75, 1.75)^{\mathrm{T}}$ 开始能找到的最近的近似解，误差精度为 10^{-8}。不难推导，我们可以得到以下不动点迭代：

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 3 - x_2 \\ \sqrt{\dfrac{11 - 3x_1^2}{2}} \end{pmatrix} \tag{7.45}$$

如果每次迭代时右端都是用上一次迭代的值，则也可称之为 Jacobi 迭代：

$$\begin{pmatrix} x_1^{(k+1)} \\ x_2^{(k+1)} \end{pmatrix} = \begin{pmatrix} 3 - x_2^{(k)} \\ \sqrt{\dfrac{11 - 3(x_1^{(k)})^2}{2}} \end{pmatrix} \tag{7.46}$$

如果我们在更新 $x_2^{(k+1)}$ 时使用的是 $x_1^{(k+1)}$ 而不是 $x_1^{(k)}$，则我们也可称之为 Gauss-Seidel 迭代：

$$\begin{pmatrix} x_1^{(k+1)} \\ x_2^{(k+1)} \end{pmatrix} = \begin{pmatrix} 3 - x_2^{(k)} \\ \sqrt{\dfrac{11 - 3(x_1^{(k+1)})^2}{2}} \end{pmatrix} \tag{7.47}$$

我们以 $(0.75, 1.75)^{\mathrm{T}}$ 为初始值，发现 Jacobi 迭代到 131 步时收敛，而 Gauss-Seidel 迭代到 62 步时收敛。我们这里举例的非线性方程组比较特殊和简单，由于方程组中的第 1 个方程是线性的，则更方便的做法是将 $x_2 = 3 - x_1$ 代入第 2 个方程中，那么就退化为求解一个一元二次方程。不动点迭代法在很多实际例子中是发散的，所以应用并不太广泛。由于非线性问题的多解性，迭代收敛一般与所取的闭区域有关，也就是我们所说的收敛域。感兴趣的读者可以查阅关于压缩映射原理和局部收敛原理的资料。如果 $\boldsymbol{x}_*$ 是不动点，且迭代格式的 Jacobi 矩阵在 $\boldsymbol{x}_*$ 处的值的谱半径小于 1，则能够找到一个 $\boldsymbol{x}_*$ 的邻域，在邻域内取任何初值都可以收敛到不动点。

7.2.2 Newton-Raphson 迭代

Newton-Raphson 迭代其实就是牛顿法，在前面的章节中有所介绍。和本章不同的是，之前要解决的问题是 $\nabla F = 0$，而本章要解决的问题是 $F = 0$，本质上都是一样的。不过我们这里不妨针对非线性方程组的求解把牛顿法讲得具体一些，除了方法推导，我们还会讨论它的收敛性质。

分别将方程 $f_1(x_1, \cdots, x_n), f_2(x_1, \cdots, x_n), \cdots, f_n(x_1, \cdots, x_n)$ 在第 k 次近似解 $\boldsymbol{x}^{(k)} = (x_1^{(k)}, x_2^{(k)}, \cdots, x_n^{(k)})$ 处以泰勒级数展开，只保留到一次项：

$$\begin{cases} f_1(\boldsymbol{x}) \approx f_1(\boldsymbol{x}^{(k)}) + \dfrac{\partial f_1}{\partial x_1}(x_1 - x_1^{(k)}) + \cdots + \dfrac{\partial f_1}{\partial x_n}(x_n - x_n^{(k)}) = 0 \\ f_2(\boldsymbol{x}) \approx f_2(\boldsymbol{x}^{(k)}) + \dfrac{\partial f_2}{\partial x_1}(x_1 - x_1^{(k)}) + \cdots + \dfrac{\partial f_2}{\partial x_n}(x_n - x_n^{(k)}) = 0 \\ \vdots \\ f_n(\boldsymbol{x}) \approx f_n(\boldsymbol{x}^{(k)}) + \dfrac{\partial f_n}{\partial x_1}(x_1 - x_1^{(k)}) + \cdots + \dfrac{\partial f_n}{\partial x_n}(x_n - x_n^{(k)}) = 0 \end{cases} \tag{7.48}$$

写成矩阵的形式是

$$\begin{pmatrix} \dfrac{\partial f_1}{\partial x_1} & \dfrac{\partial f_1}{\partial x_2} & \cdots & \dfrac{\partial f_1}{\partial x_n} \\ \dfrac{\partial f_2}{\partial x_1} & \dfrac{\partial f_2}{\partial x_2} & \cdots & \dfrac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial f_n}{\partial x_1} & \dfrac{\partial f_n}{\partial x_2} & \cdots & \dfrac{\partial f_n}{\partial x_n} \end{pmatrix} \begin{pmatrix} x_1^{(k+1)} - x_1^{(k)} \\ x_2^{(k+1)} - x_2^{(k)} \\ \vdots \\ x_n^{(k+1)} - x_n^{(k)} \end{pmatrix} = - \begin{pmatrix} f_1(\boldsymbol{x}^{(k)}) \\ f_2(\boldsymbol{x}^{(k)}) \\ \vdots \\ f_n(\boldsymbol{x}^{(k)}) \end{pmatrix} \tag{7.49}$$

即 $\nabla F(\boldsymbol{x}^{(k)})(\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)}) = -F(\boldsymbol{x}^{(k)})$，于是有如下迭代格式：

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - (\nabla F(\boldsymbol{x}^{(k)}))^{-1} F(\boldsymbol{x}^{(k)}) \tag{7.50}$$

还是以7.2.1节的非线性方程组为例，牛顿迭代的系数矩阵为

$$\nabla F(\boldsymbol{x}) = \begin{pmatrix} 1 & 1 \\ 6x_1 & 4x_2 \end{pmatrix} \tag{7.51}$$

它的逆为

$$(\nabla F(\boldsymbol{x}))^{-1} = \begin{pmatrix} 1 + \dfrac{6x_1}{4x_2 - 6x_1} & -\dfrac{1}{4x_2 - 6x_1} \\ \dfrac{-6x_1}{4x_2 - 6x_1} & \dfrac{1}{4x_2 - 6x_1} \end{pmatrix} \tag{7.52}$$

求解的迭代收敛过程如表7.4所示。

表 7.4 牛顿迭代解非线性方程组的迭代过程

步　数	近似解序列	$\|\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)}\|_2$
1	$(0.75, 1.75)^{\mathrm{T}}$	–
2	$(0.875, 2.125)^{\mathrm{T}}$	0.395
3	$(0.9759615, 2.0240385)^{\mathrm{T}}$	0.143
4	$(0.9987104, 2.0012896)^{\mathrm{T}}$	0.032
5	$(0.9999959, 2.0000041)^{\mathrm{T}}$	0.002
6	$(1, 2)^{\mathrm{T}}$	5.842×10^{-6}
7	$(1, 2)^{\mathrm{T}}$	6.034×10^{-11}

求平方根 $x^2 = n$ 是非常经典的一个问题。如果设定误差范围 ϵ，那么它的二分法可以写为算法45的形式，迭代次数是 $\log \frac{n}{\epsilon}$，每次迭代中都有乘法操作，复杂度为 $F(d)$，d 为要计算的数字有效位数，所以采用二分法求平方根的复杂度为 $\mathcal{O}(F(d) \log \frac{n}{\epsilon})$。$F(d)$ 依赖于计算两个 d 位有效数字乘法所选取的具体方法，比如用 Karatsuba 算法时，$F(d) \approx \mathcal{O}(d^{1.585})$；

用 k-way Toom Cook 算法时，$F(d) \approx \mathcal{O}(d^{\log(2k-1)/\log k})$，当 k 很大时，$\log(2k-1)/\log k$ 接近于 1。那么我们现在关心的是，如果使用牛顿法，则它的时间复杂度比二分法又快多少呢？

采用二分法求平方根问题，如算法45所示。

算法 45 采用二分法求平方根问题

输入： $\epsilon > 0$，$n > 0$

1: low $\leftarrow 0$, high $\leftarrow \max(1, n)$
2: **while** $|\text{low} - \text{high}| > \epsilon$ **do**
3: $\quad \text{mid} \leftarrow \text{low} + \dfrac{\text{high} - \text{low}}{2}$
4: $\quad$ **if** mid $\times$ mid $< n$ **then**
5: $\quad\quad$ low $\leftarrow$ mid
6: $\quad$ **else**
7: $\quad\quad$ high $\leftarrow$ mid
8: $\quad$ **end if**
9: **end while**
10: $x_* \leftarrow$ min

采用牛顿法求解平方根问题，如算法 46所示。

算法 46 采用牛顿法求平方根问题

输入： ϵ 为两次迭代近似解的误差容忍范围，n，初始解 $x_0 \leftarrow \max(1, n)$，迭代次数 $k \leftarrow 0$，初始误差 err $\leftarrow 1$

1: **while** $|\text{err}| > \epsilon$ **do**
2: $\quad x^{(k+1)} \leftarrow x^{(k)} - \dfrac{x^{(k)} \times x^{(k)} - n}{2x^{(k)}}$;
3: $\quad \text{err} \leftarrow x^{(k+1)} - x^{(k)}$;
4: $\quad k \leftarrow k + 1$;
5: **end while**
6: $x_* \leftarrow x^{(k)}$;

设 $\epsilon^{(k)} = x^{(k)} - \sqrt{n}$，那么我们可以得到：

$$\epsilon^{(k+1)} + \sqrt{n} = \frac{(\epsilon^{(k)} + \sqrt{n})^2 + n}{2(\epsilon^{(k)} + \sqrt{n})} \tag{7.53}$$

所以

$$\begin{aligned}\epsilon^{(k+1)} &= \frac{(\epsilon^{(k)}+\sqrt{n})^2+n}{2(\epsilon^{(k)}+\sqrt{n})} - \sqrt{n} \\ &= \frac{(\epsilon^{(k)})^2}{2(\epsilon^{(k)}+\sqrt{n})} \\ &< \frac{(\epsilon^{(k)})^2}{2\sqrt{n}}\end{aligned} \tag{7.54}$$

我们考虑 n 很大的情况，当 $\epsilon^{(k)}$ 远小于 $\sqrt{n}$ 时，我们可以近似地得到 $\epsilon^{(k+1)} \approx \mathcal{O}((\epsilon^{(k)})^2)$，也就是说当迭代进行到很多步误差比较小后，牛顿法是平方收敛的。设最后需要保留的有效数字位数为 d，则迭代次数为 $\mathcal{O}(\log d)$，因为每次迭代都需要进行除法操作，而除法操作的复杂度和乘法是一样的，于是采用牛顿法求平方根的复杂度为 $F(d)\log d$。不难证明 $d \approx \log n - \log \epsilon$，则 $\log(\frac{n}{\epsilon}) = \log n - \log \epsilon > \log d$，所以牛顿法优于二分法。

7.2.3 无约束优化方法

我们在讨论线性方程组的解法时讲到，我们可以把问题转换为一个严格凸函数的无约束优化问题，对于非线性方程组，我们仍然可以采用这种方法。构造目标函数

$$\phi(\boldsymbol{x}) = \frac{1}{2}\sum_{i=1}^{n}(f_i(\boldsymbol{x}))^2 \tag{7.55}$$

则

$$F(\boldsymbol{x}_*) = 0 \Leftrightarrow \phi(\boldsymbol{x}_*) = \min_{\boldsymbol{x}} \phi(\boldsymbol{x}) \tag{7.56}$$

在解决这个无约束优化问题时，可以参考前面章节中提到的拟牛顿法和最速下降法等。需要注意的是，新的问题并不一定而且往往不是严格凸函数，所以不容易获得全局最优解。

第8章

插值与拟合

插值与拟合都属于建模方法，根据已知数据确定函数的具体表达式。假设已知数据的 n 组为 $(X_0, Y_0), (X_1, Y_1), \cdots, (X_{n-1}, Y_{n-1})$，则插值就是要确定函数 f，使得 $f(X_0) = Y_0$，$f(X_1) = Y_1, \cdots, f(X_{n-1}) = Y_{n-1}$。而拟合则更为灵活，只需 $f(X_0) \approx Y_0, f(X_1) \approx Y_1, \cdots, f(X_{n-1}) \approx Y_{n-1}$。请注意这里的“约等于”并不是严格的表述方式，一般会根据实际情况定义估价函数，使得函数 f 大致符合这种近似关系。

通过概念上的阐述，我们可以推断插值与拟合的不同应用场景。应当选择插值还是拟合，这需要根据实际问题来确定。例如数据量的大小（即 n 的大小）经常起到决定性的作用：当数据量很小时，每一组数据都至关重要，所以一般采用插值；当数据量很大时，每一组数据的重要性不大，而更重要的是这些数据总体所表现出来的趋势，所以一般采用拟合。无论是采用插值还是拟合，其根本目的都是一致的：根据已知数据推断未知数据。而计算插值或者拟合的过程就相当于让函数 f 学习已知数据，掌握其中的规律，用于预测。

在本章中我们将介绍几种插值与拟合的常用方法，并使用图像处理中的一些实际问题举例说明。

8.1 插值

假定区间 $[a, b]$ 上的实值函数 $f(x)$ 在该区间上的 n 个互不相同点 $x_0, x_1, \cdots, x_{n-1}$ 处的值是 $f(x_0), f(x_1), \cdots, f(x_{n-1})$，要求估算 $f(x)$ 在 $[a, b]$ 中某点 x^* 的值，这时可以采用插值法。它的基本思想是找到一个插值函数 $P(x)$，在以上给定的离散点上与 $f(x)$ 的值相等，于

是我们可以把 $P(x^*)$ 的值作为对 $f(x^*)$ 的预测。如果 x^* 在给定点的区间内，则为内插法；反之为外插法。插值函数 $P(x)$ 的不同选取办法对应不同的插值算法，本节我们会介绍最近邻插值、线性插值、多项式插值、牛顿插值和埃尔米特插值。在实际应用中，还有很多其他广泛应用的插值算法，例如样条插值及三角插值，这里我们不做介绍，感兴趣的读者可以查阅文献来了解更多的这类算法。

8.1.1 常见的插值算法

1. 最近邻插值

最近邻插值可以说是最简单的插值方法。它的思想就是，对未知的数据从已知数据中找到距离最近的一个点来代替，即

$$P(X) = Y_{\arg\min_{1 \leqslant i \leqslant n} \|X - X_i\|} \tag{8.1}$$

式(8.1)看着复杂，实际上非常简单。这个算法也非常容易推广到二维和三维的情况，以二维为例，当我们知道矩形中 4 个点的值时，那么矩形内任意一点的值可以拿离它最近的顶点的值近似，也就是把矩形分成“田”字形，每一小矩形内所有点的值都近似为包含它的大矩形顶点的值。这种方法操作简单，无须复杂计算，但插值函数不平滑，没有连续性。可以想象如果拿它作为图像插值算法，则很容易出现锯齿形物体边界。

2. 线性插值

如果知道 $f(x)$ 在两个点 x_0、x_1 上的值，那么对于 $\forall x \in (x_0, x_1)$，我们可以采用线性函数来近似。这个线性函数表示为：

$$P(x) = f(x_0) + \frac{x - x_0}{x_1 - x_0}(f(x_1) - f(x_0)) \tag{8.2}$$

它的计算也非常简单，而且具有连续性，不足之处是误差在一般情况下随着插值点的距离的增大，插值的近似效果会变差很多。例如，原函数是 $f(x) = x^2 - 100$，如果我们取插值点为 $x_0 = -10, x_1 = 10$，那么原函数在两点都取值为 0，采用线性插值时可以得到 $P(x)$ 在区间内的所有点都近似为 0。当插值空间是二维时，将每个方向单独考虑，分别使用线性插值，则对应的方法叫作双线性插值，这也是非常自然的一种高维推广方式。

3. 多项式插值

多项式插值可以用于估计非常复杂的曲线。当已知 n 个数据点的时候，我们可以确定带有 n 个变量的多项式。记 $f(x)=a_0+a_1x+\cdots+a_{n-1}x^{n-1}$，则有

$$\begin{aligned}
a_0+a_1x_0+\cdots+a_{n-1}x_0^{n-1}&=y_0\\
a_0+a_1x_1+\cdots+a_{n-1}x_1^{n-1}&=y_1\\
&\vdots\\
a_0+a_1x_{n-1}+\cdots+a_{n-1}x_{n-1}^{n-1}&=y_{n-1}
\end{aligned}\tag{8.3}$$

请注意在这里只有 $a_0,a_1,\cdots,a_{n-1}$ 是变量。因此上式又可以写为

$$\begin{pmatrix}
1 & x_0 & \cdots & x_0^{n-1}\\
1 & x_1 & \cdots & x_1^{n-1}\\
\vdots & \vdots & & \vdots\\
1 & x_{n-1} & \cdots & x_{n-1}^{n-1}
\end{pmatrix}
\begin{pmatrix}
a_0\\ a_1\\ \vdots\\ a_{n-1}
\end{pmatrix}=
\begin{pmatrix}
y_0\\ y_1\\ \vdots\\ y_{n-1}
\end{pmatrix}\tag{8.4}$$

这是一个 n 阶线性方程组。当插值点各不相同时，可以证明该方程总是有唯一解的。

由于解方程的计算量比较大，因此我们可以采用一些特殊技巧直接构造满足要求的插值函数。例如，如果我们能构造函数 $l_0(x),l_1(x),\cdots,l_{n-1}(x)$，使得它们满足 $l_i(x_i)=1$ 且 $l_i(x_j)=0(i\neq j)$，则可以验证 $f(x)=\sum_{i=0}^{n-1}y_il_i(x)$ 刚好满足插值条件。这里 $l_i(x)$ 就是拉格朗日基，其具体表达式为

$$l_i(x)=\frac{\prod_{j\neq i}(x-x_j)}{\prod_{j\neq i}(x_i-x_j)}\tag{8.5}$$

我们也把这种采用拉格朗日基的多项式插值算法叫作拉格朗日插值算法。它的计算比直接矩阵求解方程组要方便很多，但有一个问题是当插值节点增减时，全部的插值基函数均要随之变化，整个公式也将发生变化，这在实际计算中是很不方便的。

另一种常见的插值方法是牛顿插值。我们记 n 个插值点的牛顿插值多项式为 $N_{n-1}(x)$。牛顿插值试图找到 $N_n(x)$ 与 $N_{n-1}(x)$ 的区别，以便在增加插值点的时候，从已有的插值函数出发进行修正。我们先观察 $N_n(x)-N_{n-1}(x)$，很显然 $N_n(x)-N_{n-1}(x)$ 在 $x_0,x_1,\ldots,x_{n-1}$ 处都等于 0，因此我们可以构造

$$N_n(x)-N_{n-1}(x)=b_n(x-x_0)(x-x_1)\cdots(x-x_{n-1})\tag{8.6}$$

其中 b_n 是一个常数，于是有

$$\begin{aligned} N_n(x) &= N_{n-1}(x) + b_n(x-x_0)(x-x_1)\cdots(x-x_{n-1}) \\ &= N_{n-2}(x) + b_{n-1}(x-x_0)(x-x_1)\cdots(x-x_{n-2}) \\ &\quad + b_n(x-x_0)(x-x_1)\cdots(x-x_{n-1}) \\ &= N_0(x) + b_1(x-x_0) + \cdots + b_{n-1}(x-x_0)(x-x_1)\cdots(x-x_{n-2}) \\ &\quad + b_n(x-x_0)(x-x_1)\cdots(x-x_{n-1}) \\ &= b_0 + b_1(x-x_0) + \cdots + b_n(x-x_0)(x-x_1)\cdots(x-x_{n-1}) \end{aligned} \tag{8.7}$$

我们以 3 个插值点的牛顿插值多项式为例，计算 b_0、b_1、b_2 的表达式。我们知道 $N_2(x)$ 和 $f(x)$ 在插值点 x_0、x_1、x_2 处都相同，即

$$f(x_0) = b_0 \tag{8.8}$$

$$f(x_1) = b_0 + b_1(x_1 - x_0) \tag{8.9}$$

$$f(x_2) = b_0 + b_1(x_2 - x_0) + b_2(x_2 - x_0)(x_2 - x_1) \tag{8.10}$$

由式 (8.8) 直接得到

$$b_0 = f(x_0) \tag{8.11}$$

由式(8.9)减去式(8.8)得到

$$b_1 = \frac{f(x_1) - f(x_0)}{x_1 - x_0} \tag{8.12}$$

由式(8.10)减去式(8.9)得到

$$f(x_2) - f(x_1) = b_1(x_2 - x_1) + b_2(x_2 - x_0)(x_2 - x_1) \tag{8.13}$$

从而

$$\begin{aligned} b_2 &= \frac{\dfrac{f(x_2) - f(x_1)}{x_2 - x_1} - b_1}{x_2 - x_0} \\ &= \frac{\dfrac{f(x_2) - f(x_1)}{x_2 - x_1} - \dfrac{f(x_1) - f(x_0)}{x_1 - x_0}}{x_2 - x_0} \end{aligned} \tag{8.14}$$

式(8.12)和式(8.14)称作差商。其定义可以使用新的符号表示为

$$\begin{cases} f[x_i] = f(x_i), & \forall i \\ f[x_i, \ldots, x_j] = \dfrac{f[x_{i+1}, \ldots, x_j] - f[x_i, \ldots, x_{j-1}]}{x_j - x_i}, & \forall j > i \end{cases} \tag{8.15}$$

差商表示例如表 8.1所示。

表 8.1 差商表示例

	0 阶差商	1 阶差商	2 阶差商	$\cdots$	$k-1$ 阶差商
x_0	$f[x_0]$				
x_1	$f[x_1]$	$f[x_0, x_1]$			
x_2	$f[x_2]$	$f[x_1, x_2]$	$f[x_0, x_1, x_2]$		
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	
x_k	$f[x_k]$	$f[x_{k-1}, x_k]$	$f[x_{k-2}, x_{k-1}, x_k]$	$\cdots$	$f[x_0, \cdots, x_k]$

线性插值其实就是多项式插值的一种特例。多项式插值虽然比线性插值计算更复杂，但是在一般情况下它的精度也更高。但是提到多项式插值，又不得不提到高阶多项式插值带来的 Runge 现象，这种现象的出现使得我们不能一味地靠加密插值点和使用更高阶多项式来得到更好的近似。在图 8.1中显示了不同数量的插值点对于 $f(x) = \dfrac{1}{1+x^2}$ 的近似。其中“F”代表原函数，“F4”代表使用 4 个插值点，“F6”代表使用 6 个插值点，“F8”代表使用 8 个插值点。可以发现随着插值点个数的增加，震荡现象更加明显。

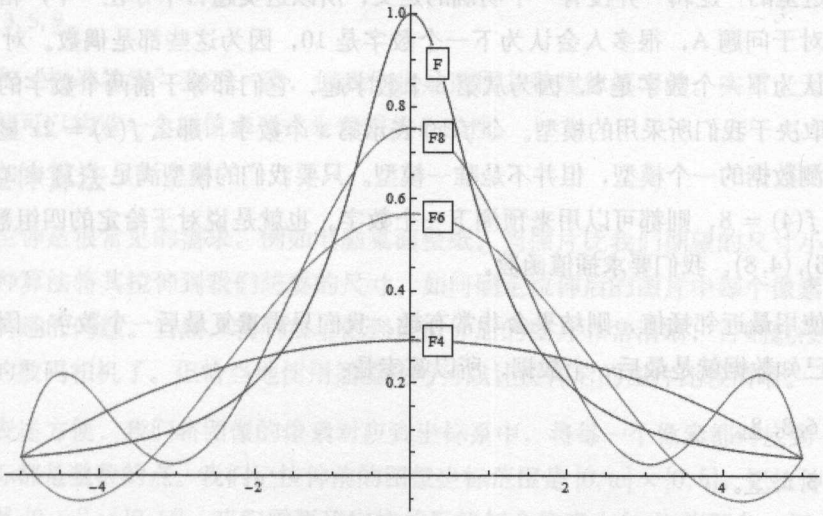


图 8.1 多项式插值的 Runge 现象图示

以上介绍的是由 n 个点确定 1 个 $n-1$ 次多项式的插值方法，它只能保证插值函数在指定点与原函数相等，在实际应用中我们可能还需要保证插值函数在指定点的一阶或者直到 k 阶的导数都是相等的，这时可以采用埃尔米特（Hermite）插值方法。Hermite 插值在不同节点满足的插值条件的个数可以不等，若在某节点 x_i 要求插值函数多项式的函数值，以及从一阶导数值直到 $k-1$ 阶导数值均与原函数相等，则我们称 x_i 为 k 重插值节点。所以，Hermite 插值除了给出插值点 $\{x_i\}_{i=0}^n$，还会给出对应的重数 $\{m_i\}_{i=0}^n$。当重数之和为 $n-1$ 时，说明插值条件有 n 个，此时 Hermite 插值多项式对应的是一个 n 次多项式。常用的二次埃尔米特插值法要求在各个插值点的值及一阶导数的值都等于原函数，此时如果有 n 个点，那么埃尔米特插值多项式为 $2n-1$ 次。其基本思想也是先确定函数形式，然后根据插值条件确定系数。

8.1.2 插值的应用

1. “逻辑测试”问题

有一些“逻辑测试”的题目，就是列出几个已知数字，让我们来推测后面的数字。例如有这样两个问题：

(A) 2, 4, 6, 8, ____

(B) 1, 2, 3, 5, ____

由于这里的“逻辑”并没有一个明确的定义，所以这类题目不存在一个严格正确的答案。例如对于问题 A，很多人会认为下一个数字是 10，因为这些都是偶数。对于问题 B，很多人会认为下一个数字是 8，因为从第 3 个数字起，它们都等于前两个数字的和。实质上，答案取决于我们所采用的模型。令 $f(x)$ 表示第 x 个数字，那么 $f(x)=2x$ 显然是符合问题 A 观测数据的一个模型，但并不是唯一模型。只要我们的模型满足 $f(1)=2, f(2)=4, f(3)=6, f(4)=8$，则都可以用来预测下一个数字。也就是说对于给定的四组数据 $(1,2), (2,4), (3,6), (4,8)$，我们要求插值函数。

如果使用最近邻插值，则结果会非常有趣。我们只需重复最后一个数字，因为距离最近的一个已知数据就是最后一个数据，所以答案是：

(A) 2, 4, 6, 8, 8。

(B) 1, 2, 3, 5, 5。

使用多项式插值后，前面的两个问题就会分别变为求解线性方程组

$$
\begin{aligned}
a_0 + a_1 + a_2 + a_3 &= 2 \\
a_0 + 2a_1 + 4a_2 + 8a_3 &= 4 \\
a_0 + 3a_1 + 9a_2 + 27a_3 &= 6 \\
a_0 + 4a_1 + 16a_2 + 64a_3 &= 8
\end{aligned}
\tag{8.16}
$$

和

$$
\begin{aligned}
a_0 + a_1 + a_2 + a_3 &= 1 \\
a_0 + 2a_1 + 4a_2 + 8a_3 &= 2 \\
a_0 + 3a_1 + 9a_2 + 27a_3 &= 3 \\
a_0 + 4a_1 + 16a_2 + 64a_3 &= 5
\end{aligned}
\tag{8.17}
$$

这两个方程组只有右端项是不同的。我们可以解出问题 A 的插值多项式为

$$
f(x) = 2x \tag{8.18}
$$

问题 B 的插值多项式为

$$
f(x) = -1 + \frac{17}{6}x - x^2 + \frac{1}{6}x^3 \tag{8.19}
$$

根据这个插值多项式，我们应当这样回答这个问题：

(A) 2, 4, 6, 8, 10。

(B) 1, 2, 3, 5, 9。

虽然和“标准答案”会不一致，但我们也是遵照某种规律填写的。实质上无论填写任何数字，都可以构造一个插值多项式来表明其合理性。

2. 图像拉伸算法

图像拉伸是很常见的需求，例如电脑桌面壁纸。当照片比我们期望的尺寸小时，我们就需要某种算法将其拉伸到我们想要的尺寸。如何确定拉伸后的图片中每个像素点的颜色就是一个有趣的问题。当然，我们很难做到让拉伸后的图片非常清晰，否则就没有必要追求高像素的数码相机了。但恰当地使用插值技巧可以让拉伸后的图片比较清晰。

为了表述方便，我们将图像的像素对应到坐标系中，将每一个像素都对应到一个整点，即横纵坐标都是整数的点。我们记拉伸前的图像坐标范围是 $[0, w] \times [0, h]$，记拉伸后的图像坐标范围是 $[0, w'] \times [0, h']$，我们需要确定拉伸后的每个像素 (x', y') 的颜色，记 (x', y') 在拉伸前的图像中对应的坐标是 (x, y)。很显然，它们之间的对应关系是

$$\frac{x}{w}=\frac{x'}{w'},\frac{y}{h}=\frac{y'}{h'} \tag{8.20}$$

当且仅当 (x,y) 是整点时，我们才明确地知道该位置的颜色。而当 (x',y') 是整点时，它所对应的 $(\frac{wx'}{w'},\frac{hy'}{h'})$ 并不一定是整点。

如果采用最近邻插值算法，则我们可以找到与其最近的整点，使用该位置的颜色代替，因此得到

$$f(x,y)=f(\left\lfloor\frac{wx'}{w'}+0.5\right\rfloor,\left\lfloor\frac{hy'}{h'}+0.5\right\rfloor) \tag{8.21}$$

另一种常用的方案是双线性插值算法。双线性插值允许选择 4 组已知数据，因此我们可以选择与 $(\frac{wx'}{w'},\frac{hy'}{h'})$ 最近的 4 个整点，即

$$\begin{aligned}(x_0,y_0)&=(\lfloor x\rfloor,\lfloor y\rfloor)\\(x_0,y_1)&=(\lfloor x\rfloor,\lceil y\rceil)\\(x_1,y_0)&=(\lceil x\rceil,\lfloor y\rfloor)\\(x_1,y_1)&=(\lceil x\rceil,\lceil y\rceil)\end{aligned} \tag{8.22}$$

可以把这 4 个点分成两组，每组的两个点仅有一维是不同的，分别对这两组进行线性插值。(x_0,y_0) 和 (x_0,y_1) 之间的线性插值是

$$f(x_0,y)=f(x_0,y_0)+(y-y_0)\frac{f(x_0,y_1)-f(x_0,y_0)}{y_1-y_0} \tag{8.23}$$

同理，(x_1,y_0) 和 (x_1,y_1) 之间的线性插值是

$$f(x_1,y)=f(x_1,y_0)+(y-y_0)\frac{f(x_1,y_1)-f(x_1,y_0)}{y_1-y_0} \tag{8.24}$$

再对这两个插值函数计算线性插值，并代入前面的插值结果，得到

$$\begin{aligned}f(x,y)&=f(x_0,y)+(x-x_0)\frac{f(x_1,y)-f(x_0,y)}{x_1-x_0}\\&=\frac{(x_1-x)(y_1-y)}{(x_1-x_0)(y_1-y_0)}f(x_0,y_0)+\frac{(x_1-x)(y-y_0)}{(x_1-x_0)(y_1-y_0)}f(x_0,y_1)\\&+\frac{(x-x_0)(y_1-y)}{(x_1-x_0)(y_1-y_0)}f(x_1,y_0)+\frac{(x-x_0)(y-y_0)}{(x_1-x_0)(y_1-y_0)}f(x_1,y_1)\end{aligned} \tag{8.25}$$

另外一种得到双线性插值表达式的途径是解方程。可以令

$$f(x,y) = a_{0,0} + a_{1,0}x + a_{0,1}y + a_{1,1}xy \tag{8.26}$$

这里有 4 个未知系数，将已知的 4 组数据代入，得到

$$\begin{pmatrix} 1 & x_0 & y_0 & x_0y_0 \\ 1 & x_0 & y_1 & x_0y_1 \\ 1 & x_1 & y_0 & x_1y_0 \\ 1 & x_1 & y_1 & x_1y_1 \end{pmatrix} \begin{pmatrix} a_{0,0} \\ a_{1,0} \\ a_{0,1} \\ a_{1,1} \end{pmatrix} = \begin{pmatrix} f(x_0,y_0) \\ f(x_0,y_1) \\ f(x_1,y_0) \\ f(x_1,y_1) \end{pmatrix} \tag{8.27}$$

我们可以解出系数，从而得到双线性插值的表达式。虽然形式不同，但它与式 (8.25) 是一致的。

除了双线性插值算法，还有使用更多的已知数据的双三次插值算法。令

$$f(x,y) = \sum_{i=0}^{3}\sum_{j=0}^{3} a_{i,j}x^i y^j \tag{8.28}$$

这里有 16 个未知系数。我们采用类似 Hermite 插值的思想，对 4 个位置的函数值及导数值进行限制。对于 4 个位置处的函数值，有

$$\begin{aligned} f(x_0,y_0) &= \sum_{i=0}^{3}\sum_{j=0}^{3} a_{i,j}x_0^i y_0^j \\ f(x_0,y_1) &= \sum_{i=0}^{3}\sum_{j=0}^{3} a_{i,j}x_0^i y_1^j \\ f(x_1,y_0) &= \sum_{i=0}^{3}\sum_{j=0}^{3} a_{i,j}x_1^i y_0^j \\ f(x_1,y_1) &= \sum_{i=0}^{3}\sum_{j=0}^{3} a_{i,j}x_1^i y_1^j \end{aligned} \tag{8.29}$$

对于 4 个位置在 x 方向上的偏导数，有

$$
\begin{aligned}
f_x(x_0,y_0) &= \sum_{i=0}^{3}\sum_{j=0}^{3} a_{i,j} i x_0^{i-1} y_0^j \\
f_x(x_0,y_1) &= \sum_{i=0}^{3}\sum_{j=0}^{3} a_{i,j} i x_0^{i-1} y_1^j \\
f_x(x_1,y_0) &= \sum_{i=0}^{3}\sum_{j=0}^{3} a_{i,j} i x_1^{i-1} y_0^j \\
f_x(x_1,y_1) &= \sum_{i=0}^{3}\sum_{j=0}^{3} a_{i,j} i x_1^{i-1} y_1^j
\end{aligned}
\tag{8.30}
$$

同理，对于 4 个位置在 y 方向上的偏导数，有

$$
\begin{aligned}
f_y(x_0,y_0) &= \sum_{i=0}^{3}\sum_{j=0}^{3} a_{i,j} j x_0^{i} y_0^{j-1} \\
f_y(x_0,y_1) &= \sum_{i=0}^{3}\sum_{j=0}^{3} a_{i,j} j x_0^{i} y_1^{j-1} \\
f_y(x_1,y_0) &= \sum_{i=0}^{3}\sum_{j=0}^{3} a_{i,j} j x_1^{i} y_0^{j-1} \\
f_y(x_1,y_1) &= \sum_{i=0}^{3}\sum_{j=0}^{3} a_{i,j} j x_1^{i} y_1^{j-1}
\end{aligned}
\tag{8.31}
$$

最后，对于 4 个位置的二阶混合偏导数，有

$$
\begin{aligned}
f_{xy}(x_0,y_0) &= \sum_{i=0}^{3}\sum_{j=0}^{3} a_{i,j} i j x_0^{i-1} y_0^{j-1} \\
f_{xy}(x_0,y_1) &= \sum_{i=0}^{3}\sum_{j=0}^{3} a_{i,j} i j x_0^{i-1} y_1^{j-1} \\
f_{xy}(x_1,y_0) &= \sum_{i=0}^{3}\sum_{j=0}^{3} a_{i,j} i j x_1^{i-1} y_0^{j-1} \\
f_{xy}(x_1,y_1) &= \sum_{i=0}^{3}\sum_{j=0}^{3} a_{i,j} i j x_1^{i-1} y_1^{j-1}
\end{aligned}
\tag{8.32}
$$

这样，我们总共得到了 16 个方程，刚好可以用来求解 16 个系数。但是以上各个方程

中的左端项其实只有 4 个函数值 $f(x_0,y_0)$、$f(x_0,y_1)$、$f(x_1,y_0)$、$f(x_1,y_1)$ 是已知的，而所有导数的值都不是已知的。在实际计算中，我们可以简单地根据周边的像素近似地估计导数的值，这就是所谓的有限差分法（Finite Difference）。例如 x 方向上的偏导数可以近似为

$$\begin{aligned} f_x(x_0,y_0) &= f(x_0,y_0) - f(x_0-1,y_0) \\ f_x(x_0,y_1) &= f(x_0,y_1) - f(x_0-1,y_1) \\ f_x(x_1,y_0) &= f(x_1+1,y_0) - f(x_1,y_0) \\ f_x(x_1,y_1) &= f(x_1+1,y_1) - f(x_1,y_1) \end{aligned} \tag{8.33}$$

y 方向上的偏导数可以近似为

$$\begin{aligned} f_y(x_0,y_0) &= f(x_0,y_0) - f(x_0,y_0-1) \\ f_y(x_0,y_1) &= f(x_0,y_1+1) - f(x_0,y_1) \\ f_y(x_1,y_0) &= f(x_1,y_0) - f(x_1,y_0-1) \\ f_y(x_1,y_1) &= f(x_1,y_1+1) - f(x_1,y_1) \end{aligned} \tag{8.34}$$

二阶混合偏导数可以近似为

$$\begin{aligned} f_{xy}(x_0,y_0) &= f(x_0,y_0) + f(x_0-1,y_0-1) - f(x_0,y_0-1) - f(x_0-1,y_0) \\ f_{xy}(x_0,y_1) &= f(x_0,y_1+1) + f(x_0-1,y_1) - f(x_0,y_1) - f(x_0-1,y_1+1) \\ f_{xy}(x_1,y_0) &= f(x_1+1,y_0) + f(x_1,y_0-1) - f(x_1,y_0) - f(x_1+1,y_0-1) \\ f_{xy}(x_1,y_1) &= f(x_1+1,y_1+1) + f(x_1,y_1) - f(x_1,y_1+1) - f(x_1+1,y_1) \end{aligned} \tag{8.35}$$

请注意有限差分法得到的公式并不是唯一的，这里所给出的公式只是作为举例。

8.2 拟合

拟合与插值非常类似。拟合方法并不需要保证拟合函数在指定点的值等于原函数的值，只需要使得在所有给定点的误差值很小。对于某些问题，拟合比插值有明显的优势。

- 当数据非常多的时候，插值的计算量巨大。例如 n 组数组对应的插值多项式是 $n-1$ 次的，包括 n 个待定系数，需要求解 n 阶线性方程组。拟合则没有必要使用 $n-1$ 次多项式。

- 高次多项式插值容易出现 Runge 现象，预测效果反而不好。拟合一般不会选用高次多项式，得到的结果通常不会有剧烈波动，更符合实际。
- 某些问题的数据本身不保证精确性，甚至可能存在少量数据完全错误的现象。这类数据会严重影响插值的结果，但拟合对轻微扰动及少量错误并不敏感，更能体现整体趋势。

本节我们将介绍线性拟合和非线性拟合方法，其中在非线性拟合中我们会介绍拟合函数是多项式和 Logistic 函数的情况。更多的关于其他非线性函数做拟合的情况，感兴趣的读者可以查阅其他资料，我们在本节中就不一一介绍了。

8.2.1 常见的拟合算法

1. 线性拟合

如果我们要拿常数来拟合 n 个给定点，使得误差平方和最小，则不难证明 $P(x)=\dfrac{f(x_0)+f(x_1)+\cdots+f(x_{n-1})}{n}$，即取均值时最优。除去常数拟合的情况，最简单的拟合函数就是线性函数了，假设 $P(x)=ax+b$，则它的最小二乘问题可以写为：

$$\text{求}a,b\text{，使得}\sum_{i=0}^{n-1}\|P(x_i)-f(x_i)\|_2^2\text{最小。}$$

将 $P(x)=ax+b$ 代入，得到要求解的最小二乘问题展开后是求 $F(a,b)=(ax_0+b-f(x_0))^2+\cdots+(ax_{n-1}+b-f(x_{n-1}))^2$ 最小。

$$\begin{cases}\dfrac{\partial F}{\partial a} &= 2\sum\limits_{i=0}^{n-1}(ax_i+b-f(x_i))^2=0\\ \dfrac{\partial F}{\partial b} &= 2\sum\limits_{i=0}^{n-1}(ax_i+b-f(x_i))=0\end{cases}\tag{8.36}$$

即

$$\begin{cases}(\sum\limits_{i=0}^{n-1}x_i^2)a+(\sum\limits_{i=0}^{n-1}x_i)b-(\sum\limits_{i=0}^{n-1}x_if(x_i)) &= 0\\ (\sum\limits_{i=0}^{n-1}x_i)a+nb-(\sum\limits_{i=0}^{n-1}f(x_i)) &= 0\end{cases}\tag{8.37}$$

可以得到

$$\begin{cases} a = \dfrac{n(\sum\limits_{i=0}^{n-1} x_i f(x_i)) - (\sum\limits_{i=0}^{n-1} x_i)(\sum\limits_{i=0}^{n-1} f(x_i))}{n(\sum\limits_{i=0}^{n-1} x_i^2) - (\sum\limits_{i=0}^{n-1} x_i)^2} \\ b = \dfrac{(\sum\limits_{i=0}^{n-1} x_i^2)(\sum\limits_{i=0}^{n-1} f(x_i)) - (\sum\limits_{i=0}^{n-1} x_i f(x_i))(\sum\limits_{i=0}^{n-1} x_i)}{n(\sum\limits_{i=0}^{n-1} x_i^2) - (\sum\limits_{i}^{n-1} x_i)^2} \end{cases} \tag{8.38}$$

其实这个问题等价于求解以下关于 a, b 变量的方程组的最小二乘解：

$$\begin{cases} ax_0 + b &= f(x_0) \\ ax_1 + b &= f(x_1) \\ &\vdots \\ ax_{n-1} + b &= f(x_{n-1}) \end{cases} \tag{8.39}$$

其中系数矩阵 $\boldsymbol{A} = \begin{pmatrix} x_0 & 1 \\ x_1 & 1 \\ \vdots & \vdots \\ x_{n-1} & 1 \end{pmatrix}$，变量 $\boldsymbol{x} = (a, b)^{\mathrm{T}}$，右端项 $\boldsymbol{f} = (f(x_0), f(x_1), \cdots, f(x_{n-1}))^{\mathrm{T}}$。那么它的最小二乘解是 $\boldsymbol{x} = (\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A})^{-1}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{f}$。感兴趣的读者可以验证这与之前按照偏导得 0 算出来的结果是一致的。

2. 非线性拟合

最基本的非线性拟合就是多项式拟合。多项式拟合要求解的问题和多项式插值类似，同样是求解线性方程组。不同的是，方程的数量通常远远大于变量的数量，这时问题一般无解，需要求近似解。最常用的方法之一就是前面多次提到过的最小二乘法，在这里就不重复介绍了。

另外，神经网络（Neural Network）也可以看作一类特殊的非线性拟合。实际上，它是多层简单的线性或非线性拟合的叠加。在数学上，m 层神经网络可以写成

$$Y = f_{m-1}(f_{m-2}(\cdots(f_0(X))\cdots)) \tag{8.40}$$

其中每一个函数都代表神经网络中的一层。既然神经网络也是非线性拟合，则自然可以和一般非线性拟合一样，使用最小二乘法求解。但更为常用的算法是 Backpropagation 算法

(BP 算法)。BP 算法其实就是梯度下降法，只是在计算梯度时巧妙地根据多层的特点，采用了链式法则求解梯度，即

$$\frac{\partial Y}{\partial X}=\frac{\partial f_{m-1}}{\partial f_{m-2}}\frac{\partial f_{m-2}}{\partial f_{m-3}}\cdots\frac{\partial f_1}{\partial f_0}\frac{\partial f_0}{\partial X} \tag{8.41}$$

8.2.2 拟合的应用

相机拍摄出照片，实质上就是将三维世界投影到二维平面，所得到的图像会根据拍摄角度的不同而产生变形，这就是所谓的透视变换（Perspective Transformation）。图像校正就是要消除拍摄角度对照片的影响，得到拍摄对象的正面照片。图像校正有许多实际应用，例如当我们需要识别照片上所显示的文字时，如果事先对图片进行校正处理，则可以提高整个系统的准确率。

图8.2的左边是实际拍摄的一个平板电脑的照片，右边是算法处理后得到的平板电脑的正面照片。要做到这样的一个校正，我们需要先根据某些特定算法选取若干组对应点，即实拍照片中的点与其在校正照片中的点。我们通常寻找一些容易识别的点作为对应点，例如平板电脑的 4 个角及按钮等，然后在校正照片中标出它们应当对应的位置。利用若干组这样的对应点，我们通过算法拟合算出透视变换，从而进一步得到完整的校正照片。

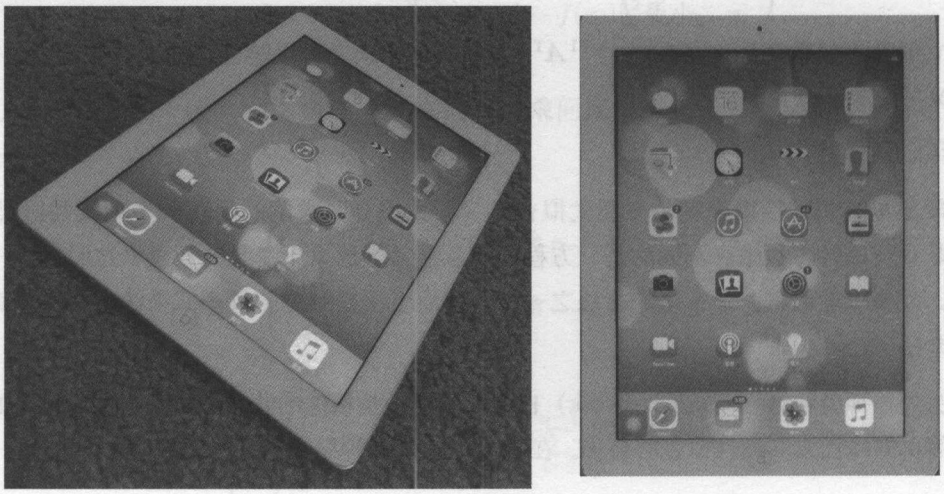


图 8.2 图像校正

假设拍摄所得的照片的某像素坐标为 (x,y)，其所对应的校正后的像素坐标为 (x',y')，那么它们之间的关系满足透视变换，具体表达式为

$$z\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} a_0 & a_1 & a_2 \\ a_3 & a_4 & a_5 \\ a_6 & a_7 & a_8 \end{pmatrix}\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} \tag{8.42}$$

因此

$$\begin{aligned} a_0x' + a_1y' + a_2 - a_6x'x - a_7y'x - a_8x = 0 \\ a_3x' + a_4y' + a_5 - a_6x'y - a_7y'y - a_8y = 0 \end{aligned} \tag{8.43}$$

我们可以发现当$a_0, a_1, \cdots, a_8$全都等于0时方程永远成立。为了避免这种情况，我们可以固定其中一个变量，不妨取$a_8 = 1$。读者可以验证，固定任何一个变量都不会影响最终结果。

每一组对应的(x, y)和(x', y')都能得到这样的两个方程，因此我们需要至少4组相互对应的像素坐标就可以根据8个方程求得透视变换。记n组像素坐标为$(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})$，以及校正后的像素坐标为$(x'_0, y'_0), (x'_1, y'_1), \cdots, (x'_{n-1}, y'_{n-1})$，则有

$$\boldsymbol{M}\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \\ a_8 \end{pmatrix} = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \\ y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} \tag{8.44}$$

其中

$$\boldsymbol{M} = \begin{pmatrix} x'_0 & y'_0 & 1 & 0 & 0 & 0 & -x'_0x_0 & -y'_0x_0 \\ x'_1 & y'_1 & 1 & 0 & 0 & 0 & -x'_1x_1 & -y'_1x_1 \\ \vdots & \vdots & 1 & 0 & 0 & 0 & \vdots & \vdots \\ x'_{n-1} & y'_{n-1} & 1 & 0 & 0 & 0 & -x'_{n-1}x_{n-1} & -y'_{n-1}x_{n-1} \\ 0 & 0 & 0 & x'_0 & y'_0 & 1 & -x'_0y_0 & -y'_0y_0 \\ 0 & 0 & 0 & x'_1 & y'_1 & 1 & -x'_1y_1 & -y'_1y_1 \\ 0 & 0 & 0 & \vdots & \vdots & 1 & \vdots & \vdots \\ 0 & 0 & 0 & x'_{n-1} & y'_{n-1} & 1 & -x'_{n-1}y_{n-1} & -y'_{n-1}y_{n-1} \end{pmatrix} \tag{8.45}$$

当 $n \geqslant 4$ 时，我们可以使用线性最小二乘法求得透视变换。另外，当 n 刚好等于 4 时，只要使用普通的解方程就可以了。对于一个给定的图像校正问题，只要对应点足够多，并且大多数都比较准确，那么最终求得的结果就会比较准确。

参考文献

[AH13] M. Alfaki and D. Haugland. Strong formulations for the pooling problem. *Journal of Global Optimization*, 56(3):897–916, 2013.

[Arn51] Walter Edwin Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quarterly of applied mathematics*, 9(1):17–29, 1951.

[BB88] J. Barzilai and J. M. Borwein. Two-point step size gradient methods. *IMA Journal of Numerical Analysis*, 8(1):141–148, 1988.

[Bel62] R. Bellman. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM (JACM)*, 9(1):61–63, 1962.

[BK73] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236, 1973.

[BL85] T. E. Baker and L. S. Lasdon. Successive linear programming at exxon. *Management science*, 31(3):264–274, 1985.

[Bro70] C. G. Broyden. The convergence of a class of double-rank minimization algorithms 1. general considerations. *IMA Journal of Applied Mathematics*, 6(1):76–90, 1970.

[Dav59] W. C. Davidon. Variable metric method for minimization. Technical report, Argonne National Lab., Lemont, Ill., 1959.

[DY99] Y. Dai and Y. Yuan. A nonlinear conjugate gradient method with a strong global convergence property. *SIAM Journal on Optimization*, 10(1):177–182, 1999.

[Fle70] R. Fletcher. A new approach to variable metric algorithms. *The computer journal*, 13(3):317–322, 1970.

[Fle13] R. Fletcher. *Practical methods of optimization*. John Wiley & Sons, 2013.

[FP63] R. Fletcher and M. J. D. Powell. A rapidly convergent descent method for minimization. *Computer journal*, 6(2):163–168, 1963.

[FR64] R. Fletcher and C. M. Reeves. Function minimization by conjugate gradients. *The computer journal*, 7(2):149–154, 1964.

[Gol70] D. Goldfarb. A family of variable-metric methods derived by variational means. *Mathematics of computation*, 24(109):23–26, 1970.

[GVL12] G. H. Golub and C. F. Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[GW95] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM (JACM)*, 42(6):1115–1145, 1995.

[Hav78] C. A. Haverly. Studies of the behavior of recursion for the pooling problem. *ACM SIGMAP Bulletin*, (25):19–28, 1978.

[Hel00] K. Helsgaun. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.

[Hir75] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[HK62] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, pages 196–210, 1962.

[HS52] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. 1952.

[HW73] C. Hierholzer and C. Wiener. Über die möglichkeit, einen linienzug ohne wiederholung und ohne unterbrechung zu umfahren. *Mathematische Annalen*, 6(1):30–32, 1873.

[KMPS03] H. Kellerer, R. Mansini, U. Pferschy, and M. G. Speranza. An efficient fully polynomial approximation scheme for the subset-sum problem. *Journal of Computer and System Sciences*, 66(2):349–370, 2003.

[Lev44] K. Levenberg. A method for the solution of certain non–linear problems in least squares. 1944.

[LK73] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.

[LWSP79] L. S. Lasdon, A. D. Waren, S. Sarkar, and F. Palacios. Solving the pooling problem using generalized reduced gradient and successive linear programming algorithms. *ACM Sigmap Bulletin*, (27):9–15, 1979.

[Mar57] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, 3(3):255–269, 1957.

[Mar63] D. W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial & Applied Mathematics*, 11(2):431–441, 1963.

[MH78] R. C. Merkle and M. E. Hellman. Hiding information and signatures in trapdoor knapsacks. *Information Theory, IEEE Transactions on*, 24(5):525–530, 1978.

[Mus97] D. R. Musser. Introspective sorting and selection algorithms. *Softw., Pract. Exper.*, 27(8):983–993, 1997.

[Noc80] J. Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of computation*, 35(151):773–782, 1980.

[NW70] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[Pol69] B. T. Polyak. The conjugate gradient method in extremal problems. *USSR Computational Mathematics and Mathematical Physics*, 9(4):94–112, 1969.

[Ros72] D. J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. *Graph theory and computing*, 183:217, 1972.

[Sha70] D. F. Shanno. Conditioning of quasi-newton methods for function minimization. *Mathematics of computation*, 24(111):647–656, 1970.

[SM02] K. U. Schulz and S. Mihov. Fast string correction with levenshtein automata. *International Journal on Document Analysis and Recognition*, 5(1):67–85, 2002.

[SS86] Y. Saad and M. H. Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3):856–869, 1986.

[TW67] W. F. Tinney and J. W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE*, 55(11):1801–1809, 1967.

[WF74] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

记录算法的点点滴滴 · 领会核心思想 · 探索工程问题

/关于本书/

本书介绍了若干常见算法，既包括排序、哈希等基础算法，也包括无约束优化、插值与拟合等数值计算方法。本书在介绍算法的同时，结合了作者自己对数学背景、应用场景的理解，便于读者把握算法的核心思想。本书尽可能地避开了以应试为导向的灌输式讲解，力求引起读者的兴趣并扩大其视野，例如在介绍哈希时，讲解了如何将哈希的算法思想运用于相似性搜索、负载均衡等多个实际问题中；又如在介绍高斯消去法时，讲解了相关的数学理论及编程实现上的具体技巧，并将其运用于对大规模稀疏线性方程组的求解，等等。

本书面向有一定高等数学及编程语言基础，对算法有初步了解的读者，包括高等院校的学生、程序员、算法分析人员及设计人员等，旨在帮助读者进一步学习算法，理解与算法相关的理论基础和应用实例。

/关于作者/

· 刁瑞，毕业于中国科学院数学与系统科学研究院，博士期间的研究方向为最优化方法。曾获2009年英特尔杯全国计算机多核程序设计大赛第1名，以及2011年KDD Cup第2名等。

· 谢妍，毕业于中国科学院数学与系统科学研究院，博士期间的研究方向为并行有限元计算。曾在微软互联网工程院从事搜索研发相关工作。

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

策划编辑：张国霞
责任编辑：徐津平
封面设计：侯士卿

欢迎投稿
邮箱：zhanggx@phei.com.cn
微信：zgx228

上架建议：计算机>算法与数据结构

ISBN 978-7-121-28671-1

定价：59.00元